8-1999

Adaptation of a long-period composite random number generator for parallel processing

Mahesh Gopalan

To the Graduate Council:

I am submitting herewith a thesis written by Mahesh Gopalan entitled "Adaptation of a long-period composite random number generator for parallel processing." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

L. Montgomery Smith, Major Professor

We have read this thesis and recommend its acceptance:

Bruce Bomar, Bruce Whitehead

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Mahesh Gopalan entitled "Adaptation of a Long Period Composite Random Number Generator for Parallel Processing." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

L Montgomery Smith

Dr. L. Montgomery Smith, Major Professor

We have read this thesis and recommend its acceptance:

[illegible]

[illegible]

Accepted for the Council:

[illegible]

Associate Vice Chancellor and
Dean of The Graduate School

Adaptation of a Long - Period Composite Random Number Generator for Parallel Processing

A thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Mahesh Gopalan
August 1999

DEDICATION

This thesis is dedicated to my parents

Mr. Govindarajalu Gopalan

and

Mrs. Prabhavathi Gopalan

who have encouraged me in all my academic endeavors.

ACKNOWLEDGMENTS

I would like to thank my major professor, Dr. Montgomery Smith for his contributions and guidance towards my research. I also wish to express my gratitude to the other members of my thesis committee, Dr. Bruce Bomar and Dr. Bruce Whitehead for their help and enthusiasm. My thanks goes to all my friends and colleagues, who have helped make my time here pleasant and memorable. Finally, I would like to thank my parents, sister, and brother-in-law, for their financial and moral support.

This work was funded by the Computational Information Processing Research Group of the University of Tennessee Space Institute under grant number E02–4001–15.

ABSTRACT

An efficient and statistically reliable random number generator is one of the most important requirements for effective Monte Carlo simulation. The latest trend in supercomputing being towards parallelization, a random number generator was designed that will allow the generation of several uncorrelated streams of random numbers in parallel.

This is achieved by the division of one period of a good serial random number generator into intervals of uniform length, one interval per processor. The serial random number generator chosen was the Marsaglia – Zaman random number generator which is a long period composite random number generator combining a linear congruential sequence with a lagged Fibonacci sequence. The mathematical relation between distantly separated seed values in each of the sequences was considered and a method was developed to obtain values from the Marsaglia – Zaman sequence spaced from each other by the length of a specified interval called the jump distance. Programs implementing the algorithm were written in C and Fortran. Seed values obtained from the programs can be used to initialize different processors to use different portions of the Marsaglia – Zaman sequence.

The seed values obtained by looping through all the random numbers in a section of the Marsaglia – Zaman sequence were shown to be identical to the seed values obtained from the developed algorithm. The execution time for the developed method was shown to increase only as the order of $\log_2$(jump distance).

TABLE OF CONTENTS

Chapter 1

Introduction

Random numbers are one of the most important components involved in computer simulations of natural phenomena using Monte Carlo methods [1]. Computer simulations find increasing use in the creation of computer models, from weather forecasting in meteorology to molecular modeling in pharmaceutical research. Possibilities for new applications open constantly. Improvements in the method of production of random numbers are therefore of particular interest to the scientific computing community.

The computation intensive applications of simulation often necessitate the use of supercomputers. The current trend in supercomputing is toward massive parallelization. Most modern supercomputers depend on the use of several processors executing in parallel to achieve very high execution speeds. The limitation of clock speeds imposed by physics on serial computers has been the most important reason for this trend. Furthermore, parallelization has become increasingly effective as it gets ever more fine-grained. For example, vector supercomputers perform one operation on multiple pieces of data concurrently, and thereby exploit the opportunities for parallelization found in most computer programs.

The efforts at the architectural level are matched by efforts at the software level. Programming methods aim at achieving faster execution by dividing tasks into smaller parts that run independently or with very little communication. Software libraries like parallel virtual machine (PVM) [2], and message passing interface (MPI) [3] have capabilities for interprocess communication that make parallel programming architecture independent. They enable a uniform distribution of tasks among the various processors. The user also strives for efficient use of processor time through maximal overlap of communication with computation. Concepts like multithreading allow parallelization at more fundamental levels, closer to the underlying operating system and the hardware.

With the increasing use of parallel programming methods, it is of interest to investigate better and more efficient parallel random number generators. The problem of generating random numbers in parallel is more involved than it is with the serial case. Not only must each individual random number stream satisfy the necessary statistical properties, but the different streams should also be uncorrelated with each other. All the numbers taken together must pass the necessary statistical tests as a whole.

The motivation for the current research was the acceleration of the Integrated Tiger Series (ITS) codes produced by Sandia National Laboratories for the simulation of radiation transport phenomena [4]. The ITS codes are under current use at the Arnold Engineering Development Center's DECADE test facility. The ITS codes find use in setting up tests and evaluating results at the test facility, which is a nuclear weapons effects simulator that tests the effects produced by intense high energy radiation on various materials. Nevertheless, the results presented here are applicable to any parallel computational code that employs random number generation. Also, the method developed is architecture independent and may be implemented on heterogeneous networks.

The ITS codes work by performing Monte Carlo random walks on various particles like photons, electrons and protons. A Monte Carlo random walk tracks the motion of a particle through a medium, given information about the various properties of the particle and of the medium that can affect the motion [5,6]. The exact path that is followed by the particle is not deterministic, and therefore probabilistic methods are used to solve the problem. This necessitates the extensive use of random numbers in the simulation. The object of this research was therefore to develop and test an efficient and statistically sound parallel random number generator that may be used in simulations such as those described above.

Current approaches toward parallel random number generation are mostly based on the generation of a different initial seed value for each processor. Each processor then runs the same serial random number generator starting with its seed. Each processor then has separate control over the execution flow. In most parallel processing systems, communication between processors is very costly. With each processor generating all the random numbers it needs, time need not be spent transporting random numbers between processors.

Most parallel random number generators are based on commonly used sequential random generators such as linear congruential generators, lagged Fibonacci generators and, feedback shift register generators. All random number generation algorithms start to repeat the same sequence beyond a certain number of values and hence cannot be used to produce more than a certain number of "random" terms. This is called the period of the sequence. One way to distribute one period of a random number sequence between n processors would be to allow the first processor to generate the first, the $(n + 1)^{th}$, the $(2n + 1)^{th}$, etc. terms of the sequence, while allowing the second processor to generate the second, the $(n + 2)^{th}$, the $(2n + 2)^{th}$, etc. terms of the sequence and so on. This procedure however slows down the process of random number generation, since the process of jumping through the random number sequence by n terms complicates most commonly used random number generating algorithms.

The most commonly adopted procedure is therefore to split a single period of a sequential generator into large equally sized sections. Each processor can then generate terms from one such section, given as a seed the starting value of that section. Thus each processor uses the same sequential random number generating algorithm, but produces independent sequences.

The problem here is one of jumping through the sequence by a very large number to generate the seeds, without having to compute all the intermediate values. This problem has been investigated for several generators, like the generalized feedback shift register generator [7], and the lagged Fibonacci generator [8], but not to the knowledge of the author, for linear congruential generators containing an additive term.

Most approaches in parallel random number generation focus on the use of a single random number generating algorithm. The approach in this research however, was to use two unrelated sequences, and then to combine values obtained by sequential progression through the two different sequences; a composite random number generator approach discussed by Marsaglia and Zaman [9]. In order to do this, it is necessary to be able to jump through both the sequences efficiently. The manner in which this is achieved is the essence of the research.

Algorithms were developed and coded in C and Fortran that could jump through the period of the composite random number generator by any arbitrary stride length. They were then tested over a sequence of length 2^{32}, and correct operation was verified by actually looping through all the random number numbers in the sequence. Some statistical tests like the uniform distribution, sequential distribution, runs-up and runs-down chi-squared tests were also performed on the random number generator.

Various sequential random number generating algorithms exist, and their applicability to parallel random number generation vary. The linear congruential generator and the lagged Fibonacci generator, which are the focus of this research are discussed in detail and the considerations involved in parallelizing them are looked at. The problem of parallelization is then tackled, and procedures are developed and implemented to solve the problem. This is followed by the execution and testing of the programs written in C and Fortran. The result is the development of a parallel random number generator that combines efficiency with statistical soundness.

Chapter 2

Background

Some of the most popular algorithms used in random number generation are linear congruential generators, lagged Fibonacci generators, multiplicative generators, shift register generators, feedback shift register generators, and cellular automata generators. The algorithms used by some of these standard random number generators are now discussed.

The most widely used of all generators are the linear congruential generators and lagged Fibonacci generators. These are looked at in detail later. Shift register generators shift the contents of a register containing the current term by a fixed number of positions, and then exclusive or the shifted value with the original value. This value may again be shifted in the opposite direction and the exclusive or operation carried out with the unshifted value to obtain the next term of the random number sequence. A feedback shift register generator is a bitwise exclusive or operator, that obtains the next bit of the sequence through the exclusive or of bits lagging the current bit by specified distances. The specific distances are obtained by finding binary primitive polynomials. Primitive polynomials of large degrees are not hard to find in the binary case, and so it is possible to obtain extremely large periods, of the order of 2^{500} with feedback shift register generators. Lastly, a cellular automata generator obtains the next term of the sequence by modifying each bit position according to a certain rule, depending on a pre-defined neighborhood of the bit.

In addition to the linear generators considered so far, non-linear generators also prove useful in many instances. An example is the multiplicative congruential generator. This type of generator multiplies two terms of the generator preceding the current term to obtain the current term. The advantage of multiplicative generators is that they pass even the most stringent statistical tests [8], but they do not cycle through all possible values for the sequence, and need a special choice of seed values.

Parallel random number generation often involves generating seeds to initialize various processes, which then proceed to use a serial random number generator to generate the necessary values. It is found that compared to using a single random number generator within each process as in most approaches, combining two very different random number generating algorithms results in a random sequence with much improved statistical properties and period.

The Marsaglia – Zaman random number generator [9] uses this property. It achieves long periods with good statistical properties by combining a linear congruential sequence with a lagged Fibonacci sequence. This causes the resulting random number sequence to have excellent statistical properties, and results in a period that is the product of the periods of the constituent generators, of the order of 2^{100}. To parallelize the Marsaglia – Zaman generator, it is necessary to find a way to skip through the constituent sequences efficiently.

Linear Congruential Generators

Linear congruential generators have the general form

$$x_{i+1} = ax_i + b \qquad (\bmod M) \qquad (2.1)$$

where a, b, x_i, and x_{i+1} are integer constants in the range $\{ 0, 1, \ldots, M-1 \}$, where the integer M is called the modulus of the generator. x_{i+1} and x_i are the $(i+1)^{\text{th}}$ and i^{th} terms of the linear congruential sequence respectively. x_0 is defined to be the seed of the sequence and must be specified before the generator can be used. The choosing of the values for a and b are dependent on ensuring the maximum possible period of recurrence for the sequence. This maximum period is bound theoretically to M. It is permissible to use values for a and b that result in a less than maximal period. However, the flexibility of obtaining the maximal period for any choice of seed will be lost in this case.

Since we desire as long a period is possible, it would be useful to make the value of M as large as possible. However, the size of M is restricted by the word size on the computer. It is possible to have M greater than what is allowed by the word size, but this would be accompanied by considerable overhead in the process of generating the random number sequence. Since efficiency is one of the prime considerations, most linear congruential generators have M less than or equal to 2^{β}, where β is the number of bits in one word. Having M exactly equal to 2^{β} results in a simplification of the implementation, since the overflow of the multiplication and addition operations automatically performs the modulus operation. Since most computing platforms have a word size of 32 bits, M may preferably be chosen equal to 2^{32}.

Another important reason to choose M to be 2^{β}, is the fact that some architectures store integers in two's complement notation. In a 32 bit computer this means, values from 0 to $2^{31} - 1$ are considered positive, while values from 2^{31} to $2^{32} - 1$ are considered negative. This difference in representation does not make a difference to the produced sequence if $M = 2^{32}$. To see how this is so, a negative number can be re-written as $x_i - 2^{32}$ where x_i would be the value of the integer in unsigned representation. Then,

$$x_{i+1} = \{ a(x_i - 2^{32}) + b \} \quad (\bmod 2^{32}) \tag{2.2}$$

$$\Rightarrow \quad x_{i+1} = \{ ax_i - 2^{32}a + b \} \quad (\bmod 2^{32}) \tag{2.3}$$

$$\Rightarrow \quad x_{i+1} = \{ ax_i + b \} \quad (\bmod 2^{32}) \tag{2.4}$$

which is the same value that would have been obtained by unsigned representation. Since portability requires the values produced by the random number generator to remain unchanged while moving from one architecture to another, the necessity of choosing $M = 2^\beta$ is apparent.

In choosing the exact values for a and b the following theorem [10] is helpful:

For $M = 2^\beta$, the generator (2.1) has full period $P = M$ if and only if,

$$a \equiv 1 \pmod{4} \text{ and } \gcd(M, b) = 1 \tag{2.5}$$

This implies b should be an odd constant, while a should be one greater than a multiple of 4. b can be chosen to be any odd constant. However, not all values of a that satisfy the above mentioned condition are acceptable. The exact value chosen for a is critically responsible for determining the statistical quality of random numbers generated. Experience and experiment are probably the only ways available to choose a value for a that results in satisfactory statistical properties. The linear congruential generator chosen for implementation in this thesis is,

$$x_{i+1} = 69069x_i + 1013904243 \qquad \pmod{2^{32}} \tag{2.6}$$

Linear congruential generators can also be designed with $b = 0$. However, these generators need a prime modulus and cannot use a value for M that equals the wordsize of the computer. This creates additional computational overhead in terms of introducing an explicit modulus operation, in place of the automatic modulus operation that happens through overflow, when M is equal to the word size of the computer. Also, when parallelizing a linear congruential generator with $b = 0$, care must be taken to not split the sequence at certain critical distances, failing which large correlations are introduced [11,12]. These considerations make it desirable to choose an odd integer value for b.

Jumping through a certain number of terms of a linear congruential sequence when $b = 0$ is not very difficult since, $x_n = a^n x_0 \pmod{M}$, and $a^n \pmod{M}$ can easily be computed as is seen later. However jumping through a linear congruential sequence that also involves an additive term introduces additional complexity, and this problem is looked at in the process of developing the parallel version of the Marsaglia – Zaman generator.

Lagged Fibonacci Generators

Lagged Fibonacci generators have the general form,

$$x_i = a_1 x_{i-1} + a_2 x_{i-2} + \dots + a_k x_{i-k} \qquad (\bmod M) \tag{2.7}$$

Here $a_k \neq 0$, and $a_1, a_2, \dots, a_k$, are integer constants that may take on both positive and negative values. M is the modulus of the addition and multiplication operations. The terms of the sequence $x_0, x_1, \dots$, can take on integer values in the range $\{ 0, 1, 2, \dots, M-1 \}$. As can be seen from the expression, it is necessary to store the last k values of the sequence at any time in order to compute the $(k + 1)$th term. The greater the number of terms of the sequence that are stored, greater is the period that can be achieved by the random number generator. The maximum possible period of a Fibonacci generator $M^k - 1$. This is because the total number of combinations that can be represented using k numbers of modulus M is given by M^k, and the combination of zeroes for all k values cannot be used.

As in the case of the linear congruential generator, we seek to choose coefficients $a_1, \dots, a_k$ such that the period of the generator is maximum possible, namely $M^k - 1$. The following theorem [13], is used to determine the coefficients:

A necessary and sufficient condition for the generator (2.7) to achieve maximal period when M is prime is that its characteristic polynomial

$$f(x) = x^k - a_1 x^{n-1} - a_2 x^{n-2} - \dots - a_{k-1} x - a_k \tag{2.8}$$

where $a_k \not\equiv 0 \ (\bmod M)$, be primitive.

In order for $f(x)$ to be primitive, it is required that the order of f be equal to $M^k - 1$. The function is said to be of order l if l is the smallest positive integer for which $f(x)$ divides $x^l - 1$. It is possible to find a primitive polynomial of degree k by generating values for $a_1, \ldots, a_k$ at random and then testing for the primitivity of the resulting function. Testing for primitivity requires computing the prime factorization of $(M^k - 1) / (M - 1)$. This problem is intractable for large k, and this poses a limitation on the number of preceding terms that may be included in the lagged Fibonacci generator.

As before, the additional benefit of choosing coefficients that result in the maximum period is the flexibility of choosing any set of seeds to initialize the random number generator. Irrespective of the choice of seed values, the period of the generator will always be maximal. It is not possible to guarantee the statistical quality of any arbitrary maximal period sequence. Experimental verification must play a role in making the final choice of coefficients for the random number generator.

Unlike the linear congruential generator that can use a value of M equal to the word size on the computer, the lagged Fibonacci generator uses a prime modulus. This is usually a prime number smaller than the word size of the computer, to simplify the computations. While the modulus operation for the linear congruential generator was automatically performed through overflow, it must be performed as a separate operation for the lagged Fibonacci generator. This means that multiple precision arithmetic is needed to handle overflow and to perform the modulus operation.This however is very computation intensive. Fortunately, it is possible to design good lagged Fibonacci generators that use only two non-zero coefficients, each of those being equal to ± 1. Also, the value of M is chosen so that its bit representation uses one less bit than the wordsize on the computer. Then for this case overflow does not occur, and the modulus operation is a one step subtraction or addition operation. One such lagged Fibonacci generator is chosen for implementatation in this thesis. It is given by,

$$x_i = x_{i-3} - x_{i-1} \qquad \bmod (2^{31}-69) \qquad (2.9)$$

In order to parallelize a lagged Fibonacci generator, it is necessary to be able to jump through the sequence by a very large value, without having to loop through the sequence. In order to achieve this, a matrix representation of the lagged Fibonacci generator is used. Then the n^{th} term of the sequence could be obtained as $\boldsymbol{x_n} = \boldsymbol{D}^n * \boldsymbol{x_0}$ (mod M), where $\boldsymbol{x_n}$ and $\boldsymbol{x_0}$ are column matrices, each containing k consecutive terms of the lagged Fibonacci sequence, with the terms in $\boldsymbol{x_n}$ succeeding the corresponding terms in $\boldsymbol{x_0}$ by n terms. $\boldsymbol{D}$ is a square matrix containing the coefficients of the lagged Fibonacci generator. For the previously chosen lagged Fibonacci generator, the matrix product is,

$$\begin{pmatrix} x_n \\ x_{n+1} \\ x_{n+2} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 2147483578 \end{pmatrix} * \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

It may be recognized that the value 2147483578 is -1 mod $(2^{31} - 69)$. The method of raising $\boldsymbol{D}$ to a very large power is described by Jun Makino[8].

Thus to achieve the parallelization of the Marsaglia – Zaman generator, it is now necessary to find a technique to be able to jump through a linear congruential sequence involving an additive term, and to implement a procedure to solve the problem. Also necessary is the implementation of a procedure to raise a given square matrix to a very large power, modulo the prime integer M.

Chapter 3

The Procedure

The linear congruential generator

The equation for the linear congruential sequence is given by equation (2.1) reproduced here for convenience,

$$x_{i+1} = ax_i + b \qquad (\bmod M) \tag{3.1}$$

Given the i^{th} term in the sequence, it is necessary to develop a procedure to find the term of the sequence that occurs after a specified number of other terms, namely the $(i + n)^{\text{th}}$ term of the sequence. It is possible to derive by analogy the n^{th} term after x_i, through consideration of the expressions for the second, third, and such terms following x_i.

$$x_{i+2} = ax_{i+1} + b \qquad (\bmod M) \tag{3.2}$$

$$\Rightarrow \quad x_{i+2} = a(ax_i + b) + b \qquad (\bmod M) \tag{3.3}$$

$$\Rightarrow \quad x_{i+2} = a^2x_i + b(a + 1) \qquad (\bmod M) \tag{3.4}$$

Similarly,

$$x_{i+3} = ax_{i+2} + b \qquad (\bmod M) \tag{3.5}$$

$$\Rightarrow \quad x_{i+3} = a(a^2x_i + b(a + 1)) + b \qquad (\bmod M) \tag{3.6}$$

$$\Rightarrow \quad x_{i+3} = a^3x_i + b(a^2 + a + 1) \qquad (\bmod M) \tag{3.7}$$

Then,

$$\Rightarrow \quad x_{i+n} = a^n x_i + b(a^{n-1} + a^{n-2} + \ldots + a + 1) \pmod{M} \tag{3.8}$$

$$\Rightarrow \quad x_{i+n} = a^n x_i + ba^n/(a-1) \pmod{M} \tag{3.9}$$

It needs to be remembered that n can be a very large value, say of the order of 2^{80}, and may even be much greater. The first problem in the evaluation of expression (3.9) is finding a^n. It would be impractical to attempt to multiply a with itself n times, and a faster way to evaluate a^n is necessary. If $b_m b_{m-1} \ldots b_0$ is the binary representation of n, then

$$a^n = a^{b_m b_{m-1} \ldots b_0} \tag{3.10}$$

$$\Rightarrow \quad a^n = a^{(2^m b_m + 2^{m-1} b_{m-1} + \ldots + 2^1 b_1 + 2^0 b_0)} \tag{3.11}$$

$$\Rightarrow \quad a^n = (\ldots (\; (a^{b_m})^2 \, a^{b_{m-1}} \,)^2 \; \ldots a^{b_1} \,)^2 \; a^{b_0} \tag{3.12}$$

It is not possible to evaluate the actual value of a^n and then compute the modulus, since the actual value of a^n is extremely large. Hence the modulus operation needs to be performed intermediate to each squaring or multiplication operation. Thus it is necessary to first multiply a^{b_m} with itself and find the modulus, then multiply the result by $a^{b_{m-1}}$ and find the modulus, then multiply the result with itself and find the modulus once more, and so on.

But in order to determine the modulus that may be used, it is necessary to consider equation (3.9), which may be rewritten as,

$$x_{i+n} = (a^n x_i) \bmod M + (ba^n/(a-1)) \bmod M \tag{3.13}$$

The addition in the expression (3.13) and in subsequent expressions is understood to be modulus M. Then expression (3.13) can be furthur expressed as,

$$x_{i+n} = ((a^n \bmod M)\, x_i) \bmod M + b\left(\frac{a^n}{a-1} \bmod M\right) \bmod M \tag{3.14}$$

$$\Rightarrow \qquad x_{i+n} = ((a^n \bmod M)\, x_i) \bmod M \; +$$

$$b\left(\frac{\left\lfloor \frac{a^n}{M(a-1)} \right\rfloor M(a-1) + (a^n \bmod M(a-1))}{a-1} \bmod M\right) \bmod M \qquad (3.15)$$

Here the expression shown within the braces $\lfloor \; \rfloor$, refers to the largest integer smaller than or equal to the enclosed expression. Then,

$$x_{i+n} = ((a^n \bmod M)\, x_i) \bmod M \; +$$

$$b\left(\frac{\left\lfloor \frac{a^n}{M(a-1)} \right\rfloor M(a-1)}{a-1} \bmod M + \frac{a^n \bmod M(a-1)}{a-1} \bmod M\right) \bmod M \qquad (3.16)$$

$$\Rightarrow \qquad x_{i+n} = ((a^n \bmod M)\, x_i) \bmod M \; +$$

$$b\left(\cancel{\left(\left\lfloor \frac{a^n}{M(a-1)} \right\rfloor M\right) \bmod M} + \frac{a^n \bmod M(a-1)}{a-1} \bmod M\right) \bmod M \qquad (3.17)$$

The part of the expression marked by the arrowhead becomes zero since an integer multiple of M modulo M is zero. Also,

$$\frac{a^n \bmod M(a-1)}{a-1} < M \qquad (3.18)$$

and so the mod M that follows may be dropped.

Hence , the expression (3.17) becomes

$$x_{i+n} = ((a^n \bmod M)\, x_i) \bmod M \; + \; b\left(\frac{a^n \bmod M(a-1)}{a-1}\right) \bmod M \qquad (3.19)$$

Hence to evaluate equation (3.9), it is necessary to calculate 2 values for a^n; one with a modulus of M usable in the first occurance of a^n and the other with a modulus of $M(a-1)$, usable in the second occurance of a^n. Then, the value of a^n obtained in the first instance is multiplied with x_i modulo M, and the value of a^n obtained in the second instance is first divided by $(a-1)$ and then multiplied by b modulo M. The two terms are then summed modulo M to obtain the $x_{i+n}{}^{th}$ term of the linear congruential sequence.

The lagged Fibonacci generator

The general form of the lagged Fibonacci generator is given by equation (2.2) reproduced here for convenience.

$$x_i = a_1 x_{i-1} + a_2 x_{i-2} + \ldots + a_k x_{i-k} \pmod M \tag{3.20}$$

This sequence may be represented in matrix form as,

$$\begin{pmatrix} x_{i-k+1} \\ x_{i-k+2} \\ x_{i-k+3} \\ \vdots \\ x_{i-1} \\ x_i \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \ddots & 0 \\ \vdots & & & & \ddots & \\ 0 & 0 & 0 & 0 & & 1 \\ a_k & a_{k-1} & a_{k-2} & a_{k-3} & \cdots & a_1 \end{pmatrix} * \begin{pmatrix} x_{i-k} \\ x_{i-k+1} \\ x_{i-k+2} \\ \vdots \\ x_{i-2} \\ x_{i-1} \end{pmatrix} \tag{3.21}$$

The equation (3.21) expresses the value of x_i in terms of the previous k values of the Fibonacci sequence. A similar matrix representation to determine the value of x_{i+1} from the previous k values can be written as,

$$\begin{pmatrix} x_{i-k+2} \\ x_{i-k+3} \\ x_{i-k+4} \\ \vdots \\ x_i \\ x_{i+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \ddots & 0 \\ \vdots & & & & \ddots & \\ 0 & 0 & 0 & 0 & & 1 \\ a_k & a_{k-1} & a_{k-2} & a_{k-3} & \cdots & a_1 \end{pmatrix} * \begin{pmatrix} x_{i-k+1} \\ x_{i-k+2} \\ x_{i-k+3} \\ \vdots \\ x_{i-1} \\ x_i \end{pmatrix} \tag{3.22}$$

It can be seen that the value of the square matrix remains unchanged in equations (3.21) and (3.22). Also, the column vector to the right on equation (3.22) is the same as the column vector to the left on equation (3.21). Substituting the value for the column vector from equation (3.21) into equation (3.22),

$$\begin{pmatrix} x_{i-k+2} \\ x_{i-k+3} \\ x_{i-k+4} \\ \vdots \\ x_i \\ x_{i+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & & 0 \\ \vdots & & & & \ddots & \\ 0 & 0 & 0 & 0 & & 1 \\ a_k & a_{k-1} & a_{k-2} & a_{k-3} & \cdots & a_1 \end{pmatrix}^2 * \begin{pmatrix} x_{i-k} \\ x_{i-k+1} \\ x_{i-k+2} \\ \vdots \\ x_{i-2} \\ x_{i-1} \end{pmatrix} \tag{3.23}$$

This procedure can be extended to jump through the sequence by any arbitrary number of terms by raising the square matrix to the particular power. Thus,

$$\begin{pmatrix} x_{i-k+n} \\ x_{i-k+n+1} \\ x_{i-k+n+2} \\ \vdots \\ x_{i+n-2} \\ x_{i+n-1} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & & 0 \\ \vdots & & & & \ddots & \\ 0 & 0 & 0 & 0 & & 1 \\ a_k & a_{k-1} & a_{k-2} & a_{k-3} & \cdots & a_1 \end{pmatrix}^n * \begin{pmatrix} x_{i-k} \\ x_{i-k+1} \\ x_{i-k+2} \\ \vdots \\ x_{i-2} \\ x_{i-1} \end{pmatrix} \tag{3.24}$$

The square matrix, which is denoted by $\boldsymbol{D}$, can be raised to a large power n in a manner analogous to the method used to raise a to the power n in the case of the linear congruential generator, that is

$$\boldsymbol{D}^n = (\ \ldots (\ (\boldsymbol{D}^{b_m})^2 \boldsymbol{D}^{b_{m-1}})^2 \ \ldots \boldsymbol{D}^{b_1})^2 \ \boldsymbol{D}^{b_0}, \tag{3.25}$$

where $b_m \, b_{m-1} \, b_{m-2} \ldots b_0$ is the binary representation of n.

The modulus operation again needs to be carried out during each matrix multiplication or squaring operation. The only modulus that is used in this case is M, which for a lagged Fibonacci generator is usually a prime number as already discussed.

Program Implementation

In the implementation of the algorithm for jumping through the linear congruential generator sequence, the primary concern is portability and not efficiency. The reason efficiency is not a prime criterion is that the procedure will be called only once for each process, at the time of invocation of the process. It is only the random number generator, which is called by the process a very large number of times in the course of problem execution, for which efficiency is important. Hence the complexity of the procedure to initialize the seed for each process does not make a significant difference to the execution time of the program.

The implementation here is on a computer with a word size of 32 bits, but it is useful to ensure that the program makes no assumptions about word size, so that portability is not a problem. Though the linear congruential generator will probably have a modulus of the form 2^{β}, it is advisable to implement the program so that any generator of any modulus could be used. Also, the program must be able to perform modulo operations with a modulus greater than the word size of the computer, because of having to calculate the value of a^n modulo $M(a - 1)$. These considerations suggest that the program be written to perform variable length arithmetic with arbitrarily large numbers and arbitrarily large modulus. The best method to achieve this is through the use of character strings, since they may be of arbitrary length and can represent arbitrarily large values. In order to achieve easy manipulation of the stored value it is helpful to store the number in its binary representation as a string composed of '0's and '1's. Though this is wasteful of memory, it makes the program simpler than would be the case if each character were used to store 8 bits of the number. This is especially true in Fortran that does not have means for efficient bit level manipulation of variable values.

Hence, all problem inputs for the program are in the form of a string of '0' and '1' characters that denote the bit representation of the input. All numerical calculations are also performed using bitwise operations on the string of bits. The advantages of string representation also extend for the lagged Fibonacci generator. Since these generators usually use a prime modulus, the modulo *M* operation is not performed automatically and hence multiplication will produce numbers greater than the word size on which the modulus operation will have to be performed through integer division. So here too it is necessary to have variable length arithmetic.

An additional restriction on the string representation is that the first character of the string should always be '1'. The program treats any string beginning with '0' as an error condition. In addition to preventing inefficiency of representation and computation, this enables the value of two numbers to be compared by string size. If both numbers have the same string size, then the bits can be compared starting from the first bit position to decide which number is greater. An outcome of this method is that the number 0 is represented as a null string.

The program derives great advantage from modularization, and the organization of specific tasks in specific modules all of which are interdependent on each other. The main tasks involved in the program involve the performance of mathematical operations on binary strings. By creating a library of functions for performing such mathematical operations, string operations can be performed in a straightforward manner.

The basic string operations to be performed constitute the following; addition with or without a modulus, multiplication with or without a modulus, subtraction, and integer division. Addition with or without a modulus can easily be performed by the same function. For the multiplication operation, the C implementation performs multiplication with or without a modulus in one function through the use of recursion. If the function is called with a modulus, it performs suitable operations and then calls itself without a modulus, and so the need for a separate function is eliminated. The Fortran implementation however needs separate functions for the two multiplication operations. An additional string comparison function that compares the numerical value of two strings and determines the larger value is needed in the process of implementing all the other functions. These functions are self sufficient for performing all the variable length arithmetic required. The code listing for the C version is listed in appendix A, and the Fortran version is in appendix B. They are also available on the world wide web at "http://www.utsi.edu/cs/parallelrng".

Program usage

The program has to be called with a statement of the form,

```
ranjump( is, os );
```

for the C version, and a statement of the form,

```
RANJUMP( IS, OS )
```

for the Fortran version. The first parameter denotes an array of k+1 string variables that contain the old seed values, and the second parameter denotes an array of k+1 string variables that contain the new seed values displaced from the old seed values by the jump distance fixed within the program.

The value of k, and the jump distance n should also be defined within the function. Other values that must be defined to describe the linear congruential generator part are the values of a, b, and the modulus M. Similarly, the values that must be defined to describe the lagged Fibonacci generator part are the values of the coefficients $a_1, a_2, \dots, a_k$, and the modulus M.

Program working

The codes are now described in detail to show the method of solution of the problem. The part of the program leaping through the linear congruential sequence is self explanatory in how a^n is computed modulo M and modulo $M(a - 1)$. The two terms of the expression (3.1) are computed as discussed, and their sum modulo M gives the required new seed value. The part of the program leaping through the lagged Fibonacci sequence initializes matrix ***D*** using the values of the coefficients a_1, a_2, ... , a_k. A separate procedure – "product" in the C version, "PRODCT" in the Fortran version – is used that performs the matrix multiplication operation modulo *M* given two square matrices of specified size *k*. Then raising the matrix ***D*** to the power *n* becomes a straightforward task. In the C version, memory for matrix ***D*** is allocated dynamically, and for a generator with lag *k*, the size of the matrix needs to be *k* by *k*. The library of functions implementing the mathematical operations are examined next. Only the C version of function and variable names are referenced in the description. The Fortran versions of these names can be inferred from the code.

The multiplication function "mult":

The "mult" function has as input the character strings containing the two operands and the modulus and provides the value of the product as output. In case the product needs to be computed without a modulus operation, a null string is passed for the modulus. Since zero can never be a valid modulus, this representation is convenient. The procedure then checks if either of the operands is zero or invalid. If so, a product value of zero is returned. Otherwise, the procedure first initializes an extra string variable "op" to hold the value of the multiplicand, and then shifts "op" one bit to the left during each step of the multiplication. Another variable "prod" holds the partial product at any stage of the multiplication. The procedure then examines each bit of the multiplier, starting from the most significant bit. If the bit is a '1', then "op" is added to "prod", otherwise "prod" remains unchanged. The variable "op" is then shifted one bit to the left. When all the bits of the multiplier have been processed, the final product is in "prod". If a modulus operation is required, the the remainder of the integer division of the product by the modulus is calculated to obtain the final result.

Of note is the fact that it is necessary to use separate string variables for internal calculation of the product, namely "prod" and for the function parameter that is used to output the product, namely "prodo". This is because if the user were to call the function with the product string variable also being one of the multiplicands, modifying the value of the product variable within the function would result in losing the value of the concerned operand that the function is using to calculate the product.

The integer division fuction "intdiv":

The integer division function, also used by the multiplication operation for calculation of the modulus, has the dividend and divisor as the inputs and the result of the integer division as the output. The "quotient" string variable is initialized to zero. The function first checks to see if the divisor is zero, and if so returns after printing an error message. If the dividend is zero, the function returns without doing anything further. Otherwise, the function stores the first *i* bits of the dividend in a variable named "sub", where *i* is the number of bits in the divisor. If "sub" is less than the divisor, then one more bit from the dividend is added at the end.

The function then subtracts the value of the divisor from "sub", and adds a '1' at the end of "quotient". The next bit from the dividend is added at the end of "sub". In the case when "sub" is a null string and next bit added from the dividend is zero, the string length of "sub" has to be readjusted back to zero. So long as the value of "sub" remains less than the dividend, additional bits from the dividend if available, are added to the end of "sub" with additional zeroes being added to the end of "quotient". If and when the value of "sub" becomes greater than the divisor, the subtraction is again carried out and the process repeated. The process ends when all the bits from the dividend have been exhausted, and consequently the value of "sub" remains less than the value of the divisor. The final value of the quotient is now available for output.

The addition function "sum":

The addition function has the two operands and the modulus of addition as the input, and the sum as the output. The modulus string is made empty when addition without a modulus is necessary. The function uses the variable "res" to store the value of the sum. The length of the sum string is initialized to be one greater than the length of the longer of the two operands. The bits are added from the least to the most significant bit of the larger operand, with the carry over bit being stored after each addition. In the C implementation, the characters '0' and '1' may be treated as integers according to the value of their internal representation, and so addition and subtraction may be directly performed with the character constants. The two functions, "CHAR" and "ICHAR" are used to achieve the same objective in Fortran. If a modulus was specified when the function was invoked, the magnitude of the summation is compared with the modulus, and if the sum is larger, the modulus is subtracted from it. An implicit assumption that the operands passed to the function are less than the modulus is justifiably made. The final result of addition is copied into the output string parameter.

The subtraction function "subt":

The subtraction function receives two operands, the first of which is assumed to be greater than or equal to the second. The difference is then passed out as the result of the subtraction. The function does not take a modulus parameter. The simplifying assumptions are made since the subtraction function is never used in any context where negative numbers need to be handled. The function initializes the length of the difference variable to the length of the first operand. The second operand is then subtracted from the first operand, from the least to the most significant bit. The borrowed bit is stored in a separate variable. At the end of the subtraction, if the borrowed bit is still '1', the function prints an error message to show the negative result.

There may still be leading zeroes in the result. These are removed and the final result is copied into the output string parameter.

The string comparison function "strmag":

The string comparison function takes two operands as input and returns a value of one if the first operand is greater than or equal to the second. Otherwise, zero is returned. The function first compares the length of the two strings to decide the greater. If the lengths of the two strings are equal, then the C version compares the strings bit by bit. The next to most significant bits are compared first, since it is already known that the most significant bits, both being '1', are the same. In case the next to most significant bits are also equal, the bits following them are compared to make the decision and so on. If both operands are equal, a value of one is returned. In the Fortran version, there is the inbuilt capacity for comparing strings, which achieves the same effect as the bit by bit comparison automatically.

This completes the description of the implementation and operation of the code, and the rationale behind the same. It is desired to test the results obtained by the implementation next, and this is continued in the next chapter.

Chapter 4

Results

The first test that must be performed is the verification of the "ranjump" procedure. This involves checking to see that the procedure actually does what it is meant to do, which is to split up a sequence of values from the Marsaglia – Zaman generator into arbitrary length sub-sequences.

The period of the Marsagia – Zaman generator is of the order of 2^{94}. This is too long a period to test, and so a part of the sequence, say of length equal to 2^{32}, is considered for the test. The usage of the program in a typical parallel programming context may need the division of the sequence into 4 or 8 or 16 sub-sequences. Three test programs were written to demonstrate this use. The first test program divided up the chosen sequence of length 2^{32} into 4 sub-sequences, the second test program divided it into 8 sub-sequences, and the third divided it into 16 sub-sequences. The initial four seeds are set to randomly chosen 32 bit string values, namely

"11100101000011101001010110110100",

"1110110001011111011001000110011",

"1110001101101100110011001110", and

"10101111001001000111010111111110".

These were then passed to "ranjump" along with the variables allocated for the return of the output seed values. The jump distance was set to

$2^{32} / 4 = 2^{30}$, for the first test program,

$2^{32} / 8 = 2^{29}$, for the second test program, and

$2^{32} / 16 = 2^{28}$, for the third test program.

As has already been discussed, changing the jump distances is only a matter of changing the single constant value that fixes the jump distance within the "ranjump" procedure. The output seeds obtained after the jump are saved and again passed to "ranjump"; this time as input seed values. The output seeds obtained after that are again saved, and this is continued three times. The initial set of seeds along with the output seeds obtained after each of three iterations could then be used to initialize four different processors, on four different subsequences of the Marsaglia – Zaman sequence. Similarly, the second and third test programs serve to simulate the operation for using the Marsaglia – Zaman generator to launch parallel programs using eight and sixteen processors respectively.

Another set of test programs are written to loop through the Marsaglia – Zaman sequence with the same initial set of four seed values. The seed values are fed to the program as the unsigned 32-bit integers 3842938292, 1982837299, 238472398 and, 2938402302. These are the integer equivalents of the binary strings listed previously. The random numbers obtained after looping through the sequence the appropriate number of times are printed as output. Correct operation is verified from the fact that the binary string values obtained at specified distances from the initial seed values through the jumping test programs, are equivalent to the respective integer values obtained through the looping programs. The binary string and integer values obtained through jumping and looping for the test cases of sequence division by four, eight and sixteen are compared in tables (4.1), (4.2) and (4.3).

The speed up achieved is demonstrated by comparing the time taken by the jumping programs, with the time taken by the looping programs in table (4.4). The time taken by the jumping program is proportional to the number of processors, since "ranjump" has to be called once for each processor. It may be recognized that even the huge differences in time taken by the jumping and looping programs are not really representative when sequence lengths much greater than 2^{32} are considered, since the time taken by the looping program will increase as the order of

Table 4.1 : Splitting a sequence of length 2^{32} into 4 equal parts

Jump distance from initial seeds	Seed values produced by jumping (Output in binary format)	Seed values produced by looping (Output in decimal format)
0 (Initial seed values)	11100101000011101001010110110100 1110110001011111011001000110011 111000110110110011001100110 10101111001001000111010111111110	3842938292 1982837299 238472398 2938402302
2^{30}	100101000011101001010110110100 10111010101001100100101001100 1111110110100111101101101110010 1000111000001010001001100001111	621712820 391432524 2127813490 1191514895
$2*2^{30}$	1100101000011101001010110110100 1110101010011001101011100101010 1000001001000100100100001101100 1101001010100010011000110100101	1695454644 1967970090 1092765804 1766928805
$3*2^{30}$	10100101000011101001010110110100 1000000011010000011101100010100 1000011101110110111100000010001 1101110110110100000111001111010	2769196468 1080572692 1136359441 1859784314

Table 4.2 : Splitting a sequence of length 2^{32} into 8 equal parts

Jump distance from initial seeds	Seed values produced by jumping (Output in binary format)	Seed values produced by looping (Output in decimal format)

0 (Initial seed values)	11100101000011101001010110110100
	1110110001011111011001000110011
	111000110110110011001100111 0
	10101111001001000111010111111110
		3842938292
		1982837299
		238472398
		2938402302

2^{29}	1010000111010010101101101 00
	1101001011010110101011111100 11
	1100000011011001011001100110100
	1010110110011100110110001000110
		84841908
		884321267
		1617736500
		1456368710

$2*2^{29}$	100101000011101001010110110100 10111010101001100100101001100 1111110110100111101101101110010 1000111000001010001001100001111	621712820 391432524 2127813490 1191514895
$3*2^{29}$	1000101000011101001010110110100 1001011100001010110110000110100 111100010001111111101110011110 110001001010011001111110000101	1158583732 1267035188 1011350430 824811397
$4*2^{29}$	1100101000011101001010110110100 1110101010011001101011100101010 1000001001000100100100001101100 1101001010100010011000110100101	1695454644 1967970090 1092765804 1766928805
$5*2^{29}$	10000101000011101001010110110100 10110010001101010000100011 110010111000101111001111010010 110001100111011001110010101011	2232325556 46715939 853734354 832412843
$6*2^{29}$	10100101000011101001010110110100 1000000011010000011101100010100 1000011101110110111100000010001 1101110110110100000111001111010	2769196468 1080572692 1136359441 1859784314

$7*2^{29}$	11000101000011101001010110110100
	1010111011110111011011
	1001011111101000100110001 10000
	1111110100011010001000101110001
		3306067380
		2866651
		1274684976
		2123174257

Table 4.3 : Splitting a sequence of length 2^{32} into 16 equal parts

Jump distance from initial seeds	Seed values produced by jumping (Output in binary format)	Seed values produced by looping (Output in decimal format)
0 (Initial seed values)	11100101000011101001010110110100 1110110001011111011001000110011 1110001101101100110011001110 10101111001001000111010111111110	3842938292 1982837299 238472398 2938402302
2^{28}	11110101000011101001010110110100 110010001111110010100100110000 1010110101100110010001000011111 1101100010101101010110101111111	4111373748 843000112 1454580255 1817619839
$2*2^{28}$	101000011101001010110110100 110100101101011010101111110011 1100000011011001011001100110100 1010110110011100110110001000110	84841908 884321267 1617736500 1456368710
$3*2^{28}$	10101000011101001010110110100 11011001111100000010111110 1000111101011110111100111101100 1110001110010100001010111010010	353277364 57131198 1202682348 1909069266
$4*2^{28}$	100101000011101001010110110100 10111010101001100100101001100 1111110110100111101101101110010 1000111000001010001001100001111	621712820 391432524 2127813490 1191514895
$5*2^{28}$	110101000011101001010110110100 10001001111111011000010100100 1100100101100000100100010110100 10111101100111010010000010010	890148276 289386660 1689274548 397648914
$6*2^{28}$	1000101000011101001010110110100 1001011100001010110110000110100 111100010001111111101110011110 110001001010011001111110000101	1158583732 1267035188 1011350430 824811397
$7*2^{28}$	1010101000011101001010110110100 1100101101101000011001110000100 1001110101011110010110010010011 1111110111001001111000111010110	1427019188 1706308484 1320103059 2128933334

Table 4.3 (Continued) :

Jump distance from initial seeds	Seed values produced by jumping (Output in binary format)	Seed values produced by looping (Output in decimal format)

$8*2^{28}$	11001010000111010010101101101 00
	11101010100110011010111001010 10
	10000010010001001001000011011 00
	11010010101000100110001101001 01
		1695454644
		1967970090
		1092765804
		1766928805

$9*2^{28}$	11101010000111010010101101101 00
	1110100001111011011001100110 11
	101100111000101101000101011 01
	110110010000111011001001111 0
		1963890100
		975100315
		376531117
		227601566

$10*2^{28}$	10000101000011101001010110110100 10110010001101010000100011 110010111000101111001111010010 110001100111011001110010101011	2232325556 46715939 853734354 832412843
$11*2^{28}$	10010101000011101001010110110100 11011100000111110001110110 1101100001011110000101001010101 10101110100110000110101111011	2500761012 57703542 1815022165 366153083
$12*2^{28}$	10100101000011101001010110110100 1000000011010000011101100010100 1000011101110110111100000010001 1101110110110100000111001111010	2769196468 1080572692 1136359441 1859784314
$13*2^{28}$	10110101000011101001010110110100 1111001001110100000011010011101 1000001100110000111010100110000 1011001010110010101110110110111	3037631924 2033845917 1100510512 1499028919
$14*2^{28}$	11000101000011101001010110110100 1010111011110111011011 1001011111110100010011000110000 1111110100011010001000101110001	3306067380 2866651 1274684976 2123174257
$15*2^{28}$	11010101000011101001010110110100 111011111001100001010011111111 11111000001101010111000111 11111011000100000000001110001	3574502836 1004934399 65066439 263258225

Table 4.4 : Comparison of execution times for jumping and looping algorithms on a Sparc station 10

Number of processors	Execution time (seconds) for jumping algorithm	Execution time (seconds) for looping algorithm
4	25	4452
8	57	5195
16	117	5593

the jump distance while the time taken by the jumping algroritthm will increase only as the order of $\log_2$(jump distance).

In order to determine the statistical quality of the random numbers, the "standard tests" of random number generation are first carried out. The standard tests include the distribution test, the serial test, and the runs up or runs down test. These tests measure the quality of the random number sequence by measuring the probability that a "perfectly random" random number generator would produce a random sequence similar to that produced by the random number generator under test. This closeness to real randomness is measured through what are called chi-squared values. A random number generator on being tested several times, will generally yield chi-squared values that have between 0.05 and 0.95 probability of being produced. Additional tests are needed to examine the statistical distribution of several of the chi-squared values produced by the random number generator.

Also, it may be noted that since many of the tests require real random numbers uniformly distributed in the range [0,1], the 32-bit integers are converted to real form by a transformation of the form,

Real_random_number = 0.5 + 2.3283064365e-10 * Integer_random_number

if the integer is signed, or a transformation of the form

Real_random_number = 2.3283064365e -10 * Integer_random_number

if the integer is unsigned. The statistical tests show a satisfactory performance on the part of the developed random number generator. Details on how the statistical tests are carried out and how the results obtained from the tests are interpreted are discussed in appendix C.

The battery of tests called DIEHARD, developed by Marsaglia [14], was also used to test the Marsaglia – Zaman generator. The results obtained along with brief descriptions about the tests, are also presented in Appendix C. The generator performs well on almost every test except the DNA test, which suggests that the pairs of bits (32,31) , (31,30) , … , (19,18) may be mutually correlated. Considering the stringent nature of this set of tests, the performance of the generator on the whole is satisfactory.

Chapter 5

Conclusions

It was desired to develop an efficient and statistically sound parallel random number generator that could be used in applications as Monte Carlo simulation. The possibility of using the Marsaglia – Zaman generator, and developing it for parallel application, was considered. It was realized that the problem of parallelization involved parallelizing the two components of the Marsaglia – Zaman generator, namely the linear congruential component and the lagged Fibonacci component, both of which were discussed in detail.

The mathemetical relations connecting terms distantly seperated in these sequences were analyzed. This enabled the development of a mechanism to jump through the two sequences. The problem of implementation and data representation was addressed and the chief objective of portability was achieved. Both C and Fortran code were written for the task. Also, the program was written to allow easy modification of the used linear congruential and lagged Fibonacci generators. Any linear congruential or lagged Fibonacci generator with any number of coefficients and any modulus could be used simply by changing a few constant parameters declared at the start of the program. Jump distance can also be similarly altered.

Proper operation of the developed jumping algorithm was tested and the large improvement in execution time compared to actually looping through the sequence was demonstrated for a sub-sequence of the Marsaglia – Zaman generator of length 2^{32}. It was also seen that the proportional increase in execution time with jump distance for the looping program makes looping impossible for practical seed separation distances of the order of 2^{90}. Only the jumping algorithm with logarithmic increase in execution time is practical for such a jump distance.

Future work could attempt application of the system for jumping through linear congruential and lagged Fibonacci sequences to other random number generating algorithms based on these sequences. The exponentiation of the matrix of lagged Fibbonacci coefficients *'D'* could be speeded up by exploiting some inherent redundancies [8], thereby making the "ranjump" procedure faster. Also improving the efficiency of the program through bit level manipulation may be considered, though this may require a sacrifice of portability.

Many current parallel random number generating algorithms resort to the use of a single algorithm like lagged Fibonacci or shift register generators, but with a large number of coefficients to achieve very long periods [15, 7]. These do not take advantage of the additional statistical soundness and efficiency, as well as very long periods made possible by the combination of two simple but drastically different algorithms as does the Marsaglia – Zaman generator. Also the generation of seeds for the above mentioned algorithms is usually done with a linear congruential sequence. Though this ensures seeds will not be reused until all seeds are exhausted, it does not allow for even distribution of the period among the various processors.

Hence among the various options available for parallel random number generation, the Marsaglia – Zaman parallel random number generator is presented as a useful method of generating random numbers for parallel applications.

REFERENCES

REFERENCES

[1] Malvin H. Kalos, and Paula A. Whitlock, “Monte Carlo Methods, Volume I: Basics,” pp. 1-5, *John Wiley and sons*, 1986.

[2] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam, “PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing,” *MIT Press*, 1994. Online publication available at “http://www.netlib.org/pvm3/book/pvm-book.html”

[3] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra, “MPI: The Complete Reference,” *MIT Press*, 1996. Online publication available at “http://www.netlib.org/utk/papers/mpi-book/mpi-book.html”

[4] J. A. Halbleib, R. P. Kensek, T. A. Mehlhorn, G. D. Valdez, S. M. Seltzer, and M. J. Berger, “ITS Version 3.0: The Integrated TIGER Series of coupled electron/photon Monte Carlo transport codes,” Technical Report SAND91-1634, Sandia National Laboratories, Albuquerque, NM, March 1992.

[5] Malvin H. Kalos, and Paula A. Whitlock, “Monte Carlo Methods, Volume I: Basics,” pp. 145-167, *John Wiley and sons*, 1986.

[6] George S. Fishman, “Monte Carlo Concepts, Algorithms, and Applications,” pp. 335-580, *Springer Series in Operations Research*, 1996.

[7] Istvan Deak, “Uniform random number generators for parallel computers,” *Parallel Computing*, vol. 15, no. 1-3, pp. 155-164, September 1990.

[8] Jun Makino, “Lagged-Fibonacci random number generators on parallel computers,” *Parallel Computing*, vol. 20, no. 9, pp. 1357-1367, September 1994.

[9] George Marsaglia, and Arif Zaman, “Some portable very-long-period random number generators,” *Computers in Physics*, vol. 8, no.1, pp. 117-121, January/February 1994.

[10] T. T. Hull, and A. R. Dobell, “Random number generators,” *SIAM Review*, vol. 4, 230-254.

[11] A. De Matteis, and S. Pagnutti, “Critical distances in pseudorandom sequences generated with composite moduli,” *International Journal of Computer Mathematics*, vol. 43, no. 3-4, pp. 189-196, 1992.

[12] Jack P. C. Kleijnen, and Ben Annink, “Pseudorandom number generators for supercomputers and classical computers: A practical introduction,” vol. 63, no. 1, pp. 76-85, November 25, 1992.

[13] George S. Fishman, "Monte Carlo Concepts, Algorithms, and Applications," pp. 645-648, *Springer Series in Operations Research*, 1996.

[14] George Marsaglia, "DIEHARD: a battery of tests for random number generators," available at "http://stat.fsu.edu/~geo/diehard.html"

[15] Daniel V. Pryor, Steven A. Cuccaro, Michael Mascagni, and M. L. Robinson, "Implementation of a Portable and Reproducible Parallel Pseudorandom Number Generator," *Proceedings, Supercomputing Conference*, pp. 311-319, 1994.

APPENDICES

APPENDIX A

C CODE

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <malloc.h>

void ranjump( char **is, char **os )
{
      /* Constants to be altered when using a different linear
         congruential or lagged fibonacci sequence                  */

      /* Number of coefficients of lagged Fibonacci sequence        */
      const int k = 3;

      char    /* Coefficients of linear congruential sequence       */
              a[256]      = "10000110111001101",
              b[256]      = "111100011011101111001101110011",

              /* Modulus of linear congruential sequence            */
              mod1[256]   = "100000000000000000000000000000000",

              /* Coefficients of lagged Fibonacci sequence. Initialize
              these in the first executable statements of the code.
              Allocate space for k coefficients by setting the first
              index of af[][] equal to k.                           */
              af[3][256],

              /* Modulus of lagged Fibonacci sequence               */
              mod2[256]   = "111111111111111111111110111011",

              /* Jump distance                                      */
              n[256]      = "10000000000000000000000000000";

      /* Variables to be used in the computation                    */

      int i, j;
      char ***D, ***G, aminus1[256], a_x_mod[256],
          a_n[256] = "1", a_power_n[256] = "1", term1[256], term2[256];

      /* Functions called                                           */

      void product ( char ***, char ***, char ***, char *, int );
      void mult    ( char *, char *, char *, char *             );
      void intdiv  ( char *, char *, char *                     );
      void subt    ( char *, char *, char *                     );
      void sum     ( char *, char *, char *, char *             );

      /* Initialization for coefficients of lagged Fibonacci
         generator                                                  */
      strcpy ( af[0], "1" );
      strcpy ( af[1], ""  );
      strcpy ( af[2], "111111111111111111111110111010" );

      /* Jumping through the Linear Congruential sequence           */
```

```
i = 0;
subt ( aminus1, a, "1" );
mult ( a_x_mod, aminus1, mod1, "" );

while ( i < (int)(strlen(n) - 1) ) {
if ( n[i] == '1' )
{
      mult ( a_power_n, a_power_n, a, mod1 );
      mult ( a_n, a_n, a, a_x_mod );
}
mult ( a_power_n, a_power_n, a_power_n, mod1 );
mult ( a_n, a_n, a_n, a_x_mod );
i++;
}

if ( n[strlen(n)-1] == '1' )
{
      mult ( a_power_n, a_power_n, a, mod1 );
      mult ( a_n, a_n, a, a_x_mod );
}

mult ( term1, a_power_n, is[0], mod1 );
subt ( a_n, a_n, "1" );
intdiv ( term2, a_n, aminus1 ); // (a^n - 1) / (a - 1)
mult ( term2, term2, b, mod1 );
sum ( os[0], term1, term2, mod1 );

/* Jumping through the lagged Fibonacci sequence              */

D = (char ***) malloc ( k * sizeof(char **) );
G = (char ***) malloc ( k * sizeof(char **) );
for ( i = 0; i < k; i++ ) {
      D[i] = (char **) malloc ( k * sizeof(char *) );
      G[i] = (char **) malloc ( k * sizeof(char *) );
}
for ( i = 0; i < k; i++ ) {
      for ( j = 0; j < k; j++ ) {
            D[i][j] = (char *) malloc ( 256 * sizeof(char) );
            G[i][j] = (char *) malloc ( 256 * sizeof(char) );
      }
}

for ( i = 0; i < k; i++ ) {
      for ( j = 0; j < k; j++ ) {
            if ( i == k-1 )
            {
                  strcpy ( D[i][j], af[j] );
            }
            else if ( j == i+1 )
            {
                  strcpy ( D[i][j], "1" );
            }
            else
            {
                  strcpy ( D[i][j], ""  );
```

```
                        }
                }
        }

        for ( i = 0; i < k; i++ ) {
                for ( j = 0; j < k; j++ ) {
                        if ( i == j )
                        {
                                strcpy ( G[i][j], "1" );
                        }
                        else
                        {
                                strcpy ( G[i][j], ""  );
                        }
                }
        }

        i = 0;
        while ( i < (strlen(n) - 1) ) {
                if ( n[i] == '1' ) product(G,G,D,mod2,k);
                product(G,G,G,mod2,k);
                i++;
        }

        if ( n[strlen(n)-1] == '1' )
                product(G,G,D,mod2,k);

        for ( i = 0; i < k; i++ ) {
                term1[0] = '\0';
                for ( j = 0; j < k; j++ ) {
                        mult ( term2, G[i][j], is[j+1], mod2 );
                        sum  ( term1, term1, term2, mod2 );
                }
                strcpy ( os[i+1], term1 );
        }

        for ( i = 0; i < k; i++ ) {
                for ( j = 0; j < k; j++ ) {
                        free ( D[i][j] );
                        free ( G[i][j] );
                }
        }
        for ( i = 0; i < k; i++ ) {
                free ( D[i] );
                free ( G[i] );
        }
        free ( D );
        free ( G );

        return;
}

void product ( char ***P, char ***M, char ***N, char *mod, int d )
{
        char ***prod, temp[256];
        int i,j,k;
        void mult( char *, char *, char *, char * );
        void sum ( char *, char *, char *, char * );

        prod = (char ***) malloc ( d * sizeof(char **) );

        for ( i = 0; i < d; i++ )
              prod[i] = (char **) malloc ( d * sizeof(char *) );

        for ( i = 0; i < d; i++ )
              for ( j = 0; j < d; j++ )
                    prod[i][j] = (char *) malloc ( 256 * sizeof(char) );

        for ( i = 0; i < d; i++ )
        {
              for ( j = 0; j < d; j++ )
              {
                    prod[i][j][0] = '\0';
                    for ( k = 0; k < d; k++ )
                    {
                          mult ( temp, M[i][k], N[k][j], mod );
                          sum  ( prod[i][j], prod[i][j], temp, mod );
                    }
              }
        }

        for ( i = 0; i < d; i++ )
              for ( j = 0; j < d; j++ )
              {
                    strcpy ( P[i][j], prod[i][j] );
                    free ( prod[i][j] );
              }

        for ( i = 0; i < d; i++ )
              free ( prod[i] );

        free ( prod );
        return;
}


void mult ( char *prodo, char *op1, char *op2, char *mod )
{
        int i,k;
        char diff[256], intprod[256], prod[256], op[256];
        void sum ( char *, char *, char *, char * );
        void subt ( char *, char *, char * );
        void intdiv ( char *, char *, char * );

        if (op1[0] == '0' || op2[0] == '0' |
           op1[0] == '\0' | op2[0] == '\0')
        {
              prodo[0] = '\0';
              return;
```

```
	}

	prod[0] = '\0';

	k = strlen (op1);
	for ( i = 0; i <= k; i++ )
		op[i] = op1[i];

	for ( i = strlen(op2) - 1; i >= 0; i-- )
	{
		if ( op2[i] == '1' )
			sum ( prod, prod, op, "" );

		op[k] = '0';
		op[++k] = '\0';
	}

	if ( mod[0] != '\0' )
	{
		intdiv ( diff, prod, mod );
		mult ( intprod, diff, mod, "" );
		subt ( prod, prod, intprod );
	}

	for ( i = 0; i <= (int)strlen(prod); i++ )
		prodo[i] = prod[i];

	return;
}

void intdiv( char *quotiento, char *dividend, char *divisor )
{
	char quotient[256];
	char sub[256];
	int i,j,q,n;
	void subt( char *, char *, char * );
	int strmagcmp( char *, char * );

	n = strlen(dividend);
	quotient[0] = '\0';
	q = 0;

	if ( divisor[0] == '\0' )
	{
		printf("Division by zero error!\n");
		return;
	}
	if ( dividend[0] == '\0' ) return;

	for ( i = 0; i < (int)strlen(divisor) ; i++)
		sub[i] = dividend[i];

	j = i;
	sub[i] = '\0';
```

```
        if (strmagcmp(sub,divisor) == 0)
        {
              sub[i] = dividend[j];
              sub[++i] = '\0';
              j++;
        }

        while (strmagcmp(sub,divisor) == 1)
        {
              subt(sub,sub,divisor);
              quotient[q] = '1';
              quotient[++q] = '\0';

              i = strlen(sub);
              sub[i] = dividend[j];
              if ( sub[0] == '0' ) i--;
              sub[++i] = '\0';
              j++;

              while ((strmagcmp(sub,divisor)==0)&&(j<=n))
              {
                    quotient[q] = '0';
                    quotient[++q] = '\0';
                    sub[i] = dividend[j];
                    if ( sub[0] == '0' ) i--;
                    sub[++i] = '\0';
                    j++;
              }
        }

        for ( i = 0; i <= (int)strlen(quotient); i++ )
              quotiento[i] = quotient[i];

        return;
}

void sum(char *reso, char *op1, char *op2, char *mod)
{
        char res[256];
        int i, j, l1, l2, l, carry;
        void subt ( char *, char *, char * );
        int strmagcmp ( char *, char * );

        if (op1[0] == '0' | op2[0] == '0')
        {
              printf("Error.\n");
              return;
        }

        l1 = strlen(op1);
        l2 = strlen(op2);
        l = (l1>l2)?l1:l2;

        res[l+1] = '\0';
```

```
carry = 0;
for ( i = l1-1, j = l2-1; (i>=0) || (j>=0); i--, j-- )
{
      if (i<0)
      {
            res[l] = op2[j] + carry;
      }
      else
      {     if (j<0)
            {
                  res[l] = op1[i] + carry;
            }
            else
            {
                  res[l] = op1[i] + op2[j] - '0' + carry;
            }
      }

      if (res[l] > '1')
      {
            res[l] -= 2;
            carry   = 1;
      }
      else
      {
            carry = 0;
      }

      l--;
}

l = (l1>l2)?l1:l2;
l++;

if ( carry == 1 )
{
      res[0] = '1';
}
else
{
      for (i=0;i<l;i++)
            res[i] = res[i+1];
}

// modulus part
if (mod[0] != '\0')
{
      if (strmagcmp(res,mod) == 1)
            subt( res, res, mod );
}

for (i=0; i <= (int)strlen(res); i++ )
      reso[i] = res[i];

return;
```

```
}

void subt ( char *reso, char *op1, char *op2 )
{
      char res[256];
      int i, j, l1, l2, borrow;

      l1 = strlen(op1);
      l2 = strlen(op2);
      res[l1] = '\0';

      if (op1[0] == '0' | op2[0] == '0' | l1 < l2)
      {
            printf("Error.\n");
            return;
      }

      borrow = 0;
      for ( i = l1-1, j = l2-1; i >= 0; i--, j-- )
      {
            if (j<0)
            {
                  res[i] = op1[i] - borrow;
            }
            else
            {
                  res[i] = ( op1[i] + '0' ) - op2[j] - borrow;
            }

            if (res[i] < '0')
            {
                  res[i] += 2;
                  borrow  = 1;
            }
            else
            {
                  borrow = 0;
            }
      }

      if ( borrow == 1 )
      {
            printf("Negative result!\n");
      }

      for ( i = 0; res[i] == '0'; i++ );

      for ( j = i; j <= (int)strlen(res); j++ )
            reso[j-i] = res[j];

      return;
}

int strmagcmp(char *one, char *two)
{
        int i;
        int l1,l2;

        l1 = strlen(one);
        l2 = strlen(two);

        if (l1>l2) return(1);
        if (l1<l2) return(0);

        for (i=0;i<l1;i++)
        {
              if (one[i] > two[i]) return(1);
              if (one[i] < two[i]) return(0);
        }

        return(1); // Equal
}
```

APPENDIX B

FORTRAN CODE

```
      SUBROUTINE RANJMP ( IS, OS )
                        CHARACTER *(*) IS(*), OS(*)

      IMPLICIT LOGICAL (A-Z)

C     Constants
C     ---------
      INTEGER K
      PARAMETER ( K = 3 )

      CHARACTER *256 A, B, MOD1, AMINUS, AXMOD, AF(K), MOD2, N
      DATA  A      / '10000110111001101 ' /,
     :      B      / '111100011011101111001101110011 ' /,
     :      MOD1   / '100000000000000000000000000000000 ' /,
     :      MOD2   / '1111111111111111111111110111011 ' /,
     :      N      / '11111111111111111111111111111111 ' /,
     :      AF(1) / '1 ' /,
     :      AF(2) / ' ' /,
     :      AF(3) / '1111111111111111111111110111010 ' /


C     Variables
C     ---------

      INTEGER I, J, LENGTH
      CHARACTER *256 D(K,K), G(K,K), STORE(K,K),
     :               APN, APOWN, TERM1, TERM2
      DATA       APN / '1 ' /,
     :           APOWN / '1 ' /

C     Linear Congruential Part
C     ------------------------

      I = 1
      CALL SUBT ( AMINUS, A, '1 ' )
      CALL MULT ( AXMOD, AMINUS, MOD1, ' ' )
      LENGTH = INDEX ( N, ' ' ) - 1

 100  IF ( I .GE. LENGTH ) GOTO 199
         IF ( N(I:I) .EQ. '1' ) THEN
            CALL MULT ( APOWN, APOWN, A, MOD1 )
            CALL MULT ( APN, APN, A, AXMOD )
         ENDIF
         CALL MULT ( APOWN, APOWN, APOWN, MOD1 )
         CALL MULT ( APN, APN, APN, AXMOD )
         I = I + 1
 188  GOTO 100
 199  IF ( N(LENGTH:LENGTH) .EQ. '1' ) THEN
         CALL MULT ( APOWN, APOWN, A, MOD1 )
         CALL MULT ( APN, APN, A, AXMOD )
      ENDIF

      CALL MULT ( TERM1, APOWN, IS(1), MOD1 )
      CALL SUBT ( APN, APN, '1 ' )
      CALL INTDIV ( TERM2, APN, AMINUS )
      CALL MULT ( TERM2, TERM2, B, MOD1 )
      CALL SUM ( OS(1), TERM1, TERM2, MOD1 )

C     Lagged Fibonacci part
C     ---------------------

      DO 299 I = 1, K
         DO 299 J = 1, K
            IF ( J .EQ. I+1 ) THEN
               D(I,J) = '1 '
            ELSE
               D(I,J) = ' '
            ENDIF
            IF ( I .EQ. K ) THEN
               D(I,J) = AF(J)
            ENDIF
 299  CONTINUE

      DO 399 I = 1, K
         DO 399 J = 1, K
            IF ( I .EQ. J ) THEN
               G(I,J) = '1 '
            ELSE
               G(I,J) = ' '
            ENDIF
 399  CONTINUE

      I = 1

 400  IF ( I .EQ. LENGTH ) GOTO 499
         IF ( N(I:I) .EQ. '1' ) THEN
            CALL PRODCT ( G, G, D, STORE, MOD2, K )
         ENDIF
         CALL PRODCT ( G, G, G, STORE, MOD2, K )
         I = I + 1
         GOTO 400
 499  CONTINUE

      IF ( N(LENGTH:LENGTH) .EQ. '1' ) THEN
         CALL PRODCT ( G, G, D, STORE, MOD2, K )
      ENDIF

      DO 599 I = 1, K
         TERM1 = ' '
         DO 699 J = 1, K
            CALL MULT ( TERM2, G(I,J), IS(J+1), MOD2 )
            CALL SUM  ( TERM1, TERM1,  TERM2,   MOD2 )
 699     CONTINUE
         OS(I+1) = TERM1
 599  CONTINUE

      END

      SUBROUTINE PRODCT ( P, M, N, PROD, MOD, D )
                CHARACTER*(*) P(*), M(*), N(*), PROD(*), MOD
               INTEGER D

      CHARACTER*256 TEMP
      INTEGER I, J, K

      DO 199 I = 1, D
         DO 199 J = 1, D
            PROD(I+(J-1)*D) = ' '
            DO 199 K = 1, D
               CALL MULT ( TEMP, M(I+(K-1)*D),
     :                           N(K+(J-1)*D), MOD )
               CALL SUM  ( PROD(I+(J-1)*D), PROD(I+(J-1)*D),
     :                     TEMP, MOD )
 199  CONTINUE

      DO 299 I = 1, D
         DO 299 J = 0, D-1
            P(I+J*D) = PROD(I+J*D)
 299  CONTINUE

      END

      SUBROUTINE MULT ( PRODO, OP1, OP2, MOD )
                      CHARACTER *(*) PRODO, OP1, OP2, MOD
      INTEGER I, K
      CHARACTER*256 DIFF, INTPRD, PROD, OP

      IF ( OP1(1:1) .EQ. ' ' .OR. OP2(1:1) .EQ. ' ' ) THEN
         PRODO = ' '
         RETURN
      ENDIF

      PROD = ' '

      K = INDEX ( OP1, ' ' )
      OP = OP1

      DO 188 I = INDEX(OP2,' ')-1, 1, -1
         IF ( OP2(I:I) .EQ. '1' ) THEN
            CALL SUM ( PROD, PROD, OP, ' ' )
         ENDIF
         OP(K:K) = '0'
         K = K + 1
 188  CONTINUE

      IF ( MOD(1:1) .NE. ' ' ) THEN
         CALL INTDIV ( DIFF, PROD, MOD )
         CALL MULT1 ( INTPRD, DIFF, MOD )
         CALL SUBT ( PROD, PROD, INTPRD )
      ENDIF

      PRODO = PROD

      END
      SUBROUTINE MULT1 ( PRODO, OP1, OP2 )
                          CHARACTER*(*) PRODO, OP1, OP2

      INTEGER I, K
      CHARACTER*256 PROD, OP

      IF ( OP1(1:1) .EQ. ' ' .OR. OP2(1:1) .EQ. ' ' ) THEN
         PRODO = ' '
         RETURN
      ENDIF

      PROD = ' '

      K = INDEX ( OP1, ' ' )
      OP = OP1

      DO 188 I = INDEX(OP2,' ')-1, 1, -1
         IF ( OP2(I:I) .EQ. '1' ) THEN
            CALL SUM ( PROD, PROD, OP, ' ' )
         ENDIF
         OP(K:K) = '0'
         K = K + 1
188   CONTINUE

      PRODO = PROD

      END

      SUBROUTINE INTDIV ( QUOTO, DIVID, DIVIS )
                 CHARACTER*(*) QUOTO, DIVID, DIVIS

      CHARACTER*256 QUOT, SUB
      INTEGER I, J, Q, N
      INTEGER STRMAG

      N = INDEX ( DIVID, ' ' ) - 1
      QUOT = ' '
      Q = 1
      SUB = ' '

      IF ( DIVIS(1:1) .EQ. ' ' ) THEN
         PRINT *, 'Division by zero error!'
         RETURN
      ENDIF

      IF ( STRMAG( DIVID, DIVIS ) .EQ. 0 ) THEN
         QUOTO = ' '
         RETURN
      ENDIF

      J = INDEX ( DIVIS, ' ' )
      SUB = DIVID ( 1 : J-1 )

      IF ( STRMAG( SUB, DIVIS ) .EQ. 0 ) THEN
         SUB(J:J) = DIVID(J:J)
```

```
        J = J + 1
     ENDIF

100  CALL SUBT( SUB, SUB, DIVIS )
        QUOT(Q:Q) = '1'
        Q = Q + 1
        I = INDEX(SUB, ' ')
        SUB(I:I) = DIVID(J:J)
        J = J + 1
        IF ( SUB(1:1) .EQ. '0' ) THEN
           SUB(1:1) = ' '
        ELSE
           I = I + 1
        ENDIF
110  IF ( STRMAG( SUB, DIVIS ) .EQ. 1 ) GOTO 100
     IF ( J .GT. N+1 ) GOTO 199
        QUOT(Q:Q) = '0'
        Q = Q + 1
        SUB(I:I) = DIVID(J:J)
        J = J + 1
        IF ( SUB(1:1) .EQ. '0' ) THEN
           SUB(1:1) = ' '
        ELSE
           I = I + 1
        ENDIF
     GOTO 110
199  CONTINUE

     QUOTO = QUOT
     END

     SUBROUTINE SUM ( RESO, OP1, OP2, MOD )
                CHARACTER*(*) RESO, OP1, OP2, MOD

     CHARACTER*256 RES
     INTEGER L1, L2, L, CARRY
     INTEGER STRMAG

     L1 = INDEX( OP1, ' ' ) - 1
     L2 = INDEX( OP2, ' ' ) - 1
     L  = MAX (L1,L2)
     RES = ' '

     CARRY = 0

100  IF ( L1 .LE. 0 .AND. L2 .LE. 0 ) GOTO 199
        IF ( L1 .LE. 0 ) THEN
           RES(L:L) = CHAR ( ICHAR (OP2(L2:L2)) + CARRY )
        ELSEIF ( L2 .LE. 0 ) THEN
           RES(L:L) = CHAR ( ICHAR (OP1(L1:L1)) + CARRY )
        ELSE
           RES(L:L) = CHAR ( ICHAR (OP1(L1:L1)) + ICHAR (OP2(L2:L2)) -
    :                        ICHAR ('0') + CARRY )
     ENDIF
```

```
      IF ( RES(L:L) .GT. '1' ) THEN
         RES(L:L) = CHAR ( ICHAR ( RES(L:L) ) - 2 )
         CARRY = 1
      ELSE
         CARRY = 0
      ENDIF

      L = L - 1
      L1 = L1 - 1
      L2 = L2 - 1

188   GOTO 100
199   IF ( CARRY .EQ. 1 ) THEN
         RES = '1' // RES
      ENDIF

      IF ( MOD(1:1) .NE. ' ' .AND. STRMAG ( RES, MOD ) .EQ. 1 ) THEN
         CALL SUBT ( RES, RES, MOD )
      ENDIF

      RESO = RES
      END

      SUBROUTINE SUBT ( RESO, OP1, OP2 )
                 CHARACTER*(*) RESO, OP1, OP2

      CHARACTER*256 RES
      INTEGER L1, L2, BORROW
      INTEGER STRMAG

      L1 = INDEX ( OP1, ' ' ) - 1
      L2 = INDEX ( OP2, ' ' ) - 1
      RES = ' '

      IF ( STRMAG ( OP1, OP2 ) .EQ. 0 ) THEN
         PRINT *, 'Error... Negative Result!'
         RETURN
      ENDIF

      BORROW = 0
100   IF ( L1 .EQ. 0 ) GOTO 199
         IF ( L2 .LE. 0 ) THEN
            RES(L1:L1) = CHAR ( ICHAR( OP1(L1:L1) ) - BORROW )
         ELSE
            RES(L1:L1) = CHAR ( ICHAR(OP1(L1:L1)) - ICHAR(OP2(L2:L2)) +
     :                          ICHAR( '0' ) - BORROW )
         ENDIF
         IF ( RES(L1:L1) .LT. '0' ) THEN
            RES(L1:L1) = CHAR ( ICHAR (RES(L1:L1)) + 2 )
            BORROW = 1
         ELSE
            BORROW = 0
         ENDIF
         L1 = L1 - 1
         L2 = L2 - 1
```

```
188 GOTO 100
199 L1 = INDEX ( RES, '1' )

    IF ( L1 .NE. 0 ) THEN
       RESO = RES(L1:)
    ELSE
       RESO = ' '
    ENDIF

    END

    INTEGER FUNCTION STRMAG ( ONE, TWO )
              CHARACTER*(*) ONE, TWO

    INTEGER L1, L2

    L1 = INDEX ( ONE, ' ' ) - 1
    L2 = INDEX ( TWO, ' ' ) - 1

    IF ( L1 .GT. L2 ) THEN
       STRMAG = 1
    ELSEIF ( L1 .LT. L2 ) THEN
       STRMAG = 0
    ELSEIF ( ONE .GE. TWO ) THEN
       STRMAG = 1
    ELSE
       STRMAG = 0
    ENDIF

    END
```

APPENDIX C

STATISTICAL TESTS

The equidistribution test:

The equidistribution test tests the uniformity with which the random numbers produced by the generator are distributed over their range. The test is implemented here as follows. The range from 0 to 1 is divided into 100 intervals of width 0.01 each. 10,000 random numbers uniformly distributed in [0,1] are generated. Each random number is counted into one of the hundred intervals by multiplying the number by a hundred and considering the integer part of the result. If the random number generator truly produces uniformly distributed numbers, then each of the 100 intervals must have a count of about 100. The chi-squared value of the distribution is then calculated using the formula,

$$\chi^2 = \frac{k}{N}\sum_{i=1}^{k}\left(x_i - \frac{N}{k}\right)^2$$

where $k = 100$ is the number of intervals, x_i is the actual count of numbers found in bin i, and $N =$ 10,000 is the total count of the random numbers generated.

The test is repeated 1000 times in the same manner with different sets of 10,000 random numbers, and the value of χ^2 is computed for each test. The mean and standard deviation of the 1000 χ^2 values thus obtained are calculated. The expected and obtained values of the mean and standard deviation with the arbitrarily chosen seed values of 1982837299, 238472398, 2938402302, 3842938292, are as follows,

<u>Mean :</u>

Expected value = 100.000

Obtained value = 99.124

<u>Standard Deviation :</u>

Expected value = 14.142

Obtained value = 13.470

The 1000 values for χ^2 are divided into 100 bins. The first bin counts the number of χ^2 values obtained that have a probability of occurrence that is 0.01 or less. The second bin counts the number of χ^2 values having a probability of occurrence between 0.01 and 0.02, and so on. Since a particular value of χ^2 may occur with equal probability in any of the equiprobable bins, we expect a uniform distribution of the χ^2 values among the bins. This distribution is plotted if figure (C.1). The *x*-axis is continuos and represents the continuously varying value of χ^2. The *y*-axis is discrete and represents the count of χ^2 values obtained in each bin. Since the expected value in each bin is 10, a graph of expected count against χ^2 would be a straight line parallel to the *x*-axis and intercepting the *y*-axis at a count of 10. The actual distribution is seen to be scattered around this mean value as is to be expected. The 1000 χ^2 values are next themselves subjected to another χ^2 test. The overall chi-squared value thus obtained is given by,

$$\chi_T^2 = \sum_{j=1}^{100} \frac{(B_j - 10)^2}{10}$$

where B_j represents the number of χ^2 values in bin *j*. The overall chi-squared value also has 99 degrees of freedom. For the particular choice of seed values made, the overall chi-squared value for the Marsaglia – Zaman generator is found to be 104.00. This is a perfectly acceptable value from a $\chi^2(99)$ distribution.

The serial test:

The serial test attempts to measure any correlation between successive numbers in the random number sequence. In other words, each random number should be perfectly independent of other random numbers in its neighborhood. 10,000 pairs of random numbers are generated and distributed in a two dimensional array of bins, with 10 bins to each dimension. Both random numbers in each pair are multiplied by 10, and the integer parts of the result give the indices of

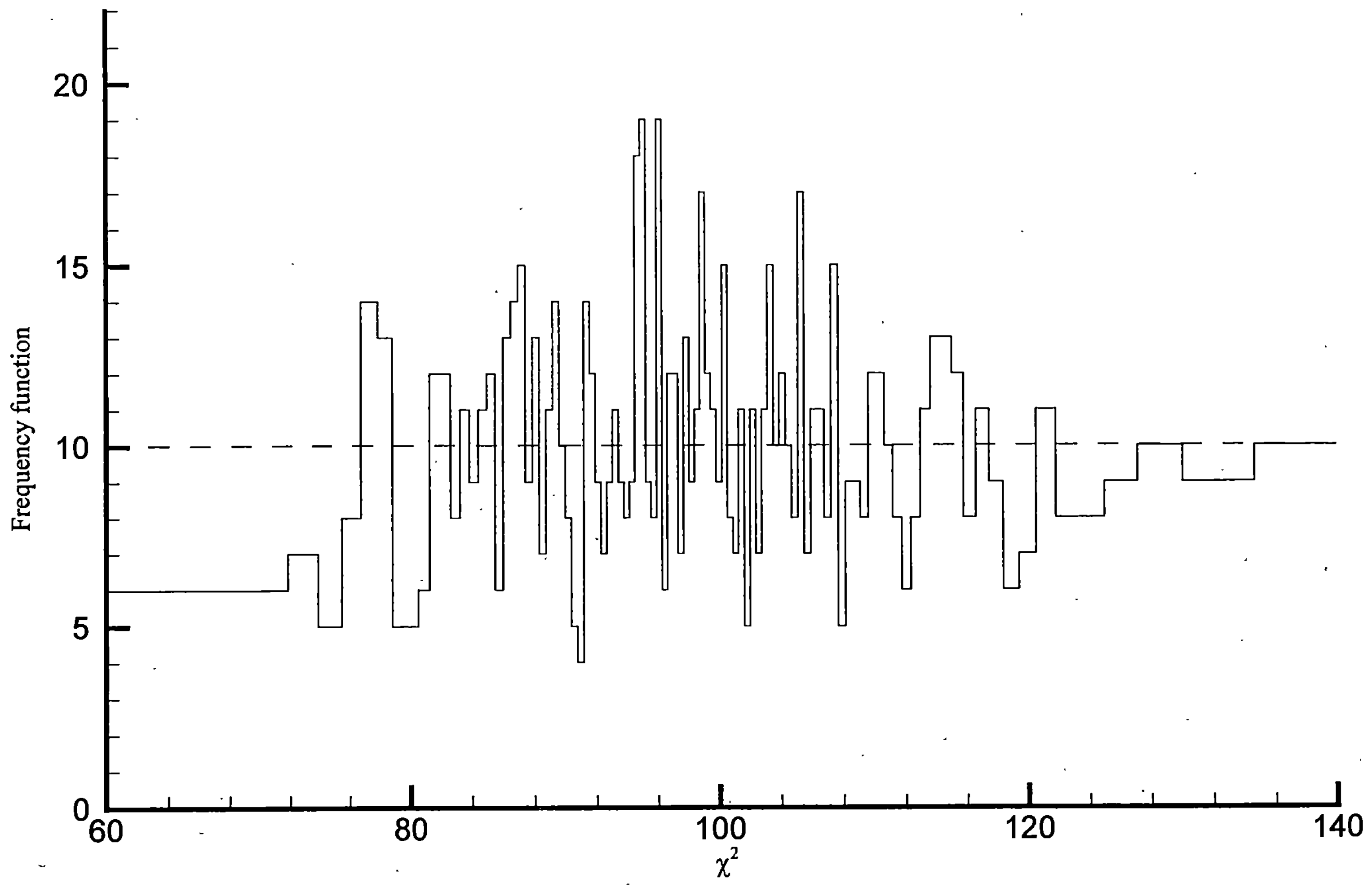

Figure (C.1) : Distribution of the 1000 χ^2 values of the equidistribution test the bin into which the pair is to be counted. The expected count in each of the 100 bins is therefore 100. The chi-squared value for the test is given by,

$$\chi^2 = \frac{k^2}{N}\sum_{i=1}^{k}\sum_{j=1}^{k}\left(x_{ij} - \frac{N}{k^2}\right)^2$$

where $k = 10$, is the number of bins in each dimension, $N = 10{,}000$ is the total count of all the bins, and x_{ij} is the observed count value in the bin with indices i and j. Though the test is applied here to two consecutive random numbers, it can be extended to larger groups of random numbers. 1000 of these tests are carried out to obtain 1000 values for χ^2. The mean and standard deviation of these 1000 values with the same choice of seed values as the equidistribution test are as follows,

Mean :

Expected value = 100.000

Observed value = 98.973

Standard Deviation :

Expected value = 14.142

Observed value = 14.036

The 1000 values are divided into 100 equiprobable bins as for the equidistribution test. The plot of the distribution against χ^2 value is shown in figure (C.2). It is seen that the distribution fluctuates around the mean value of 10 per bin as expected. An overall chi-squared value is again calculated as for the equidistribution test. This value is found to be 82.600 for the particular seeds chosen. It is a reasonable value for a sample from a $\chi^2(99)$ distribution.

Runs up and runs down test:

The runs up and runs down tests count respectively, the number of consecutively increasing or decreasing values found in the distribution. A sequence of continuously increasing or decreasing numbers is called a run, and the length of the runs can vary from one to the number

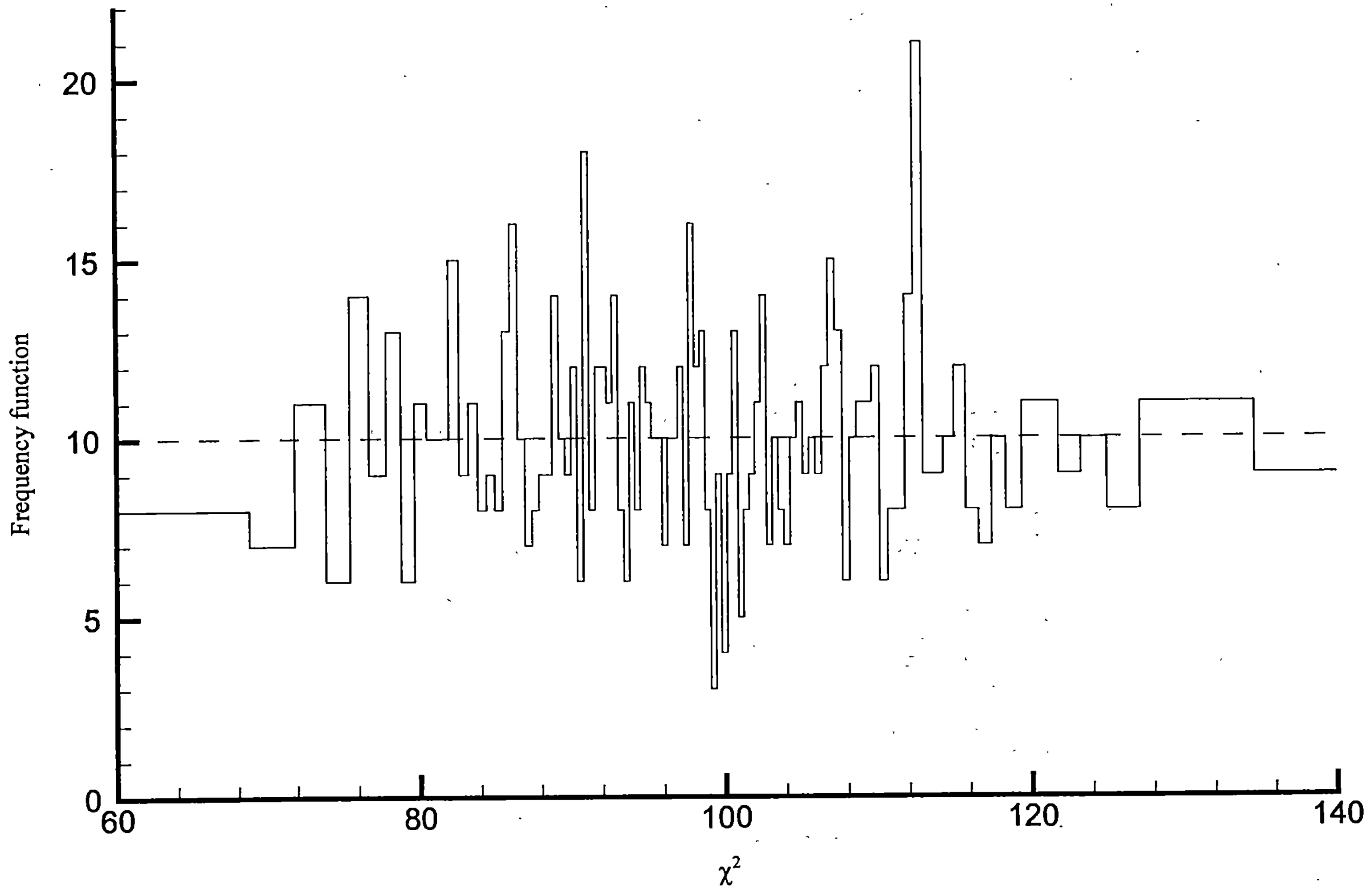

Figure (C.2) : Distribution of the 1000 χ^2 values of the serial test of random numbers in the sequence. The probability of a run of a certain length occurring can be calculated, and this helps find the expected number of runs of any given length.

The test is carried out with $N = 10{,}000$ for the Marsaglia – Zaman generator. Seed values chosen for the first run are 1982837299, 238472398, 2938402302, and 3842938292. Seed values chosen for the second run are 487113490, 1092832391, 832801293, and 1233298109. For these seed values, the number of runs up and runs down observed and expected are shown in table (C.1).

It is not possible to apply a χ^2 test directly using the estimated and obtained values of the number of runs-up and runs-down. This is because the runs are not independent of each other, and each run affects the runs adjacent to it. However, it is possible to possible to get an asymptotic chi-squared value as,

$$V = \frac{1}{N}\sum_{i=1}^{r}\sum_{j=1}^{r}(R_i - E(R_i))(R_j - E(R_j))a_{ij}$$

where R_i is the observed count of runs of length i, and $E(R_i)$ is the estimated count of runs of length i. The estimated count is given by,

$$E(R_l) = E(R'_l) - E(R'_{l+1})$$

where R'_l stands for the count of the number of runs of length l or greater. The expected value of this count is calculated using the formula,

$$E(R'_l) = \frac{(N+1)l}{(l+1)!} - \frac{l-1}{l!}$$

The terms a_{ij} stand for the elements of the inverse of the covariance matrix which is obtained by calculating the covariance of the count of the number of runs of length i, with the count of the number of runs of length j, for $1 \le i, j \le r$. Here r stands for the length of the maximum length run, all runs of length greater than r being grouped into the count for run r. In the test conducted

Table 4.1 : Runs up and runs down tests

Length of run	Number of runs-up observed	Number of runs-down observed	Expected number of runs-up / runs-down
RUN 1			
1	1689	1687	1667
2	2040	2061	2083
3	926	898	916
4	286	285	263
5	51	56	57
≥ 6	9	12	10
RUN 2			
1	1599	1662	1667
2	2142	2151	2083
3	891	916	916
4	251	241	263
5	77	47	57
≥6	9	14	10

here r is chosen to be six. The value of the elements of the covariance matrix are calculated from the expressions,

$$Cov(R_l, R'_m) = Cov(R'_l, R'_m) - Cov(R'_{l+1}, R'_m),$$
$$Cov(R_l, R_m) = Cov(R_l, R'_m) - Cov(R_l, R'_{m+1}),$$

and

$$Cov(R'_l, R'_m) = \begin{cases} E(R'_t) + f(l,m,n) : l + m \le n, \\ E(R'_t) - E(R'_l)E(R'_m) : l + m > n, \end{cases}$$

where $t = max\ \{\ l\ ,\ m\ \}$, $s = l + m$, and $f(\ l,m,n\ ) =$

$$(n+1)\left(\frac{s(1-lm)+lm}{(l+1)(m+1)} - \frac{2s}{(s+1)}\right) + 2\left(\frac{s-1}{s!}\right) + \frac{(s^2-s-2)m - s^2 - l^2m^2 + 1}{(l+1)(m+1)}$$

In calculating the covariance matrix, care should be taken to substitute R'_r in place of R_r in calculating the covariance of run r with other runs or with itself. The asymptotic chi-squared values are found to be,

Run 1 :

$V_{runs\ up} = 3.200.$ Probability = 0.217

$V_{runs\ down} = 2.253.$ Probability = 0.108

Run 2 :

$V_{runs\ up} = 7.785.$ Probability = 0.747

$V_{runs\ down} = 4.165.$ Probability = 0.344

The probability values correspond to the probability that the obtained value of V is a sample from a $\chi^2(6)$ distribution, as it is expected to be.

This completes the discussion on the standard tests. The performance of the generator has been shown to be good on all the standard tests considered. Other special and more stringent tests are also available. The only limitation in designing the special tests is the availability of a way to predict the performance of a "truly" random sequence of numbers subjected to the test, and to estimate under exactly what conditions, a random number generator under test can be said to have failed the test. One such set of stringent tests is “DIEHARD”. The results of testing the random number generator with “DIEHARD” are summarized in the following extract from the output of the program. The random numbers are provided to “DIEHARD” in a file called “ranintb.32”.

```
NOTE: Most of the tests in DIEHARD return a p-value, which
should be uniform on [0,1) if the input file contains truly
independent random bits.  Those p-values are obtained by
p=F(X), where F is the assumed distribution of the sample
random variable X---often normal. But that assumed F is just
an asymptotic approximation, for which the fit will be worst
in the tails. Thus you should not be surprised with
occasional p-values near 0 or 1, such as .0012 or .9983.
When a bit stream really FAILS BIG, you will get p's of 0 or
1 to six or more places.  By all means, do not, as a
Statistician might, think that a p < .025 or p> .975 means
that the RNG has "failed the test at the .05 level".  Such
p's happen among the hundreds that DIEHARD produces, even
with good RNG's.  So keep in mind that " p happens".

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::            This is the BIRTHDAY SPACINGS TEST                ::
:: Choose m birthdays in a year of n days.  List the spacings   ::
:: between the birthdays.  If j is the number of values that    ::
:: occur more than once in that list, then j is asymptotically  ::
:: Poisson distributed with mean m^3/(4n).  Experience shows n  ::
:: must be quite large, say n>=2^18, for comparing the results  ::
:: to the Poisson distribution with that mean.  This test uses  ::
:: n=2^24 and m=2^9,  so that the underlying distribution for j ::
:: is taken to be Poisson with lambda=2^27/(2^26)=2.  A sample  ::
:: of 500 j's is taken, and a chi-square goodness of fit test   ::
:: provides a p value.  The first test uses bits 1-24 (counting ::
:: from the left) from integers in the specified file.          ::
::   Then the file is closed and reopened. Next, bits 2-25 are  ::
:: used to provide birthdays, then 3-26 and so on to bits 9-32. ::
:: Each set of bits provides a p-value, and the nine p-values   ::
:: provide a sample for a KSTEST.                               ::
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
BIRTHDAY SPACINGS TEST, M= 512 N=2**24 LAMBDA=  2.0000
Results for ranintb.32
For a sample of size 500: using bits  1 to 24, Mean = 2.036
duplicate       number       number
spacings       observed     expected
    0            63.          67.668
    1           144.         135.335
    2           134.         135.335
    3            83.          90.224
    4            41.          45.112
    5            22.          18.045
6 to INF         13.           8.282
Chisquare with  6 d.o.f. = 5.40 p-value = .506125
p values were found similarly for tests using bits 2 to 25, 3 to 26
upto 9 to 32.
The 9 p-values were
   .506125   .220544   .513211   .412704   .025586
   .291103   .066305   .340122   .641447
A KSTEST for the 9 p-values yields  .869347
```

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::            THE OVERLAPPING 5-PERMUTATION TEST                 ::
:: This is the OPERM5 test.  It looks at a sequence of one mill- ::
:: ion 32-bit random integers.  Each set of five consecutive     ::
:: integers can be in one of 120 states, for the 5! possible or- ::
:: derings of five numbers.  Thus the 5th, 6th, 7th,...numbers   ::
:: each provide a state. As many thousands of state transitions  ::
:: are observed,  cumulative counts are made of the number of    ::
:: occurences of each state.  Then the quadratic form in the     ::
:: weak inverse of the 120x120 covariance matrix yields a test   ::
:: equivalent to the likelihood ratio test that the 120 cell     ::
:: counts came from the specified (asymptotically) normal dis-   ::
:: tribution with the specified 120x120 covariance matrix (with  ::
:: rank 99).  This version uses 1,000,000 integers, twice.       ::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
OPERM5 test for file ranintb.32
For a sample of 1,000,000 consecutive 5-tuples,
chisquare for 99 degrees of freedom=136.092; p-value= .992007
OPERM5 test for file ranintb.32
For a sample of 1,000,000 consecutive 5-tuples,
chisquare for 99 degrees of freedom= 93.783; p-value= .370658

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: This is the BINARY RANK TEST for 31x31 matrices. The leftmost ::
:: 31 bits of 31 random integers from the test sequence are used ::
:: to form a 31x31 binary matrix over the field {0,1}. The rank  ::
:: is determined. That rank can be from 0 to 31, but ranks< 28   ::
:: are rare, and their counts are pooled with those for rank 28. ::
:: Ranks are found for 40,000 such random matrices and a chisqua-::
:: re test is performed on counts for ranks 31,30,29 and <=28.   ::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
Binary rank test for ranintb.32
Rank test for 31x31 binary matrices:
rows from leftmost 31 bits of each 32-bit integer
rank   observed  expected (o-e)^2/e  sum
  28       203     211.4   .335179     .335
  29      5078    5134.0   .611052     .946
  30     22925   23103.0  1.372143    2.318
  31     11794   11551.5  5.089753    7.408
chisquare= 7.408 for 3 d. of f.; p-value= .943280

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: This is the BINARY RANK TEST for 32x32 matrices. A random 32x ::
:: 32 binary matrix is formed, each row a 32-bit random integer. ::
:: The rank is determined. That rank can be from 0 to 32, ranks  ::
:: less than 29 are rare, and their counts are pooled with those ::
:: for rank 29.  Ranks are found for 40,000 such random matrices ::
:: and a chisquare test is performed on counts for ranks  32,31, ::
:: 30 and <=29.                                                  ::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
Binary rank test for ranintb.32
Rank test for 32x32 binary matrices:
rows from leftmost 32 bits of each 32-bit integer
```

```
  rank  observed  expected (o-e)^2/e    sum
   29      231      211.4  1.813725   1.814
   30     5165     5134.0   .187059   2.001
   31    23081    23103.0   .021039   2.022
   32    11523    11551.5   .070436   2.092
chisquare= 2.092 for 3 d. of f.; p-value= .520544

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: This is the BINARY RANK TEST for 6x8 matrices.  From each of  ::
:: six random 32-bit integers from the generator under test, a   ::
:: specified byte is chosen, and the resulting six bytes form a  ::
:: 6x8 binary matrix whose rank is determined.  That rank can be ::
:: from 0 to 6, but ranks 0,1,2,3 are rare; their counts are     ::
:: pooled with those for rank 4. Ranks are found for 100,000     ::
:: random matrices, and a chi-square test is performed on        ::
:: counts for ranks 6,5 and <=4.                                  ::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
Binary Rank Test for ranintb.32
Rank of a 6x8 binary matrix,
rows formed from eight bits of the RNG ranintb.32
b-rank test for bits  1 to  8
                OBSERVED   EXPECTED     (O-E)^2/E      SUM
     r<=4          981       944.3        1.426        1.426
     r =5        21851     21743.9         .528        1.954
     r =6        77168     77311.8         .267        2.221
                   p=1-exp(-SUM/2)= .67065
The b-rank test is repeated for bits 2 to 9, 3 to 10 and so on upto
25 to 32. p is found for each test.
These should be 25 uniform [0,1] random variables:
.670646      .955596      .218858      .665258      .667737
.659341      .652131      .314140      .407926      .132801
.212144      .555781      .669843      .158414      .839626
.528217      .052615      .806658      .003447      .898659
.376459      .793129      .309774      .831839      .994243
brank test summary for ranintb.32
The KS test for those 25 supposed UNI's yields
KS p-value= .207604

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::                   THE BITSTREAM TEST                          ::
:: The file under test is viewed as a stream of bits. Call them  ::
:: b1,b2,... .  Consider an alphabet with two "letters", 0 and 1 ::
:: and think of the stream of bits as a succession of 20-letter  ::
:: "words", overlapping.  Thus the first word is b1b2...b20, the ::
:: second is b2b3...b21, and so on.  The bitstream test counts   ::
:: the number of missing 20-letter (20-bit) words in a string of ::
:: 2^21 overlapping 20-letter words.  There are 2^20 possible 20 ::
:: letter words.  For a truly random string of 2^21+19 bits, the ::
:: number of missing words j should be (very close to) normally  ::
:: distributed with mean 141,909 and sigma 428.  Thus            ::
::  (j-141909)/428 should be a standard normal variate (z score) ::
:: that leads to a uniform [0,1) p value.  The test is repeated  ::
:: twenty times.                                                  ::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

```
THE OVERLAPPING 20-tuples BITSTREAM  TEST, 20 BITS PER WORD, N
words
This test uses N=2^21 and samples the bitstream 20 times.
No. missing words should average  141909. with sigma=428.
tst no  1:  141907 missing words, -.01 sigmas from mean,
p-value=  .49783
p-values obtained by repeating the test 20 times:
.49783 .82660 .58480 .64892 .59660
.52298 .23349 .14423 .02720 .50995
.25775 .74552 .02413 .14583 .72720
.24362 .95344 .38304 .60921 .06611

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::              The tests OPSO, OQSO and DNA                   ::
::          OPSO means Overlapping-Pairs-Sparse-Occupancy      ::
:: The OPSO test considers 2-letter words from an alphabet of  ::
:: 1024 letters.  Each letter is determined by a specified ten ::
:: bits from a 32-bit integer in the sequence to be tested. OPSO ::
:: generates  2^21 (overlapping) 2-letter words  (from 2^21+1  ::
:: "keystrokes")  and counts the number of missing words---that ::
:: is 2-letter words which do not appear in the entire sequence. ::
:: That count should be very close to normally distributed with ::
:: mean 141,909, sigma 290. Thus (missingwrds-141909)/290 should ::
:: be a standard normal variable. The OPSO test takes 32 bits at ::
:: a time from the test file and uses a designated set of ten  ::
:: consecutive bits. It then restarts the file for the next de- ::
:: signated 10 bits, and so on.                                ::
::                                                             ::
::     OQSO means Overlapping-Quadruples-Sparse-Occupancy      ::
::   The test OQSO is similar, except that it considers 4-letter ::
:: words from an alphabet of 32 letters, each letter determined ::
:: by a designated string of 5 consecutive bits from the test  ::
:: file, elements of which are assumed 32-bit random integers. ::
:: The mean number of missing words in a sequence of 2^21 four- ::
:: letter words,  (2^21+3 "keystrokes"), is again 141909, with  ::
:: sigma = 295.  The mean is based on theory; sigma comes from  ::
:: extensive simulation.                                       ::
::                                                             ::
::    The DNA test considers an alphabet of 4 letters::  C,G,A,T,::
:: determined by two designated bits in the sequence of random  ::
:: integers being tested.  It considers 10-letter words, so that ::
:: as in OPSO and OQSO, there are 2^20 possible words, and the  ::
:: mean number of missing words from a string of 2^21  (over-   ::
:: lapping)  10-letter  words (2^21+9 "keystrokes") is 141909.  ::
:: The standard deviation sigma=339 was determined as for OQSO  ::
:: by simulation.  (Sigma for OPSO, 290, is the true value (to  ::
:: three places), not determined by simulation.                ::
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
OPSO test for generator ranintb.32 using bits 23 to 32
No. missing words = 142085,  Equiv normal variate = .606,
p-value = .7277
p-values obtained by repeating test using bits 22 to 31, 21 to 30
and so on upto bits 1 to 10 are:
.8707 .4954 .8317 .7918 .7808 .4529 .9077 .8509 .2693 .7898 .9995
.9411 .6874 .2278 .1008 .5489 .6029 .9473 .8729 .1354 .4216 .8386
```

```
OQSO test for generator ranintb.32 using bits 28 to 32
No. missing words = 141699, Equiv normal variate = -.713,
p-value = .2379
p-values obtained by repeating test using bits 27 to 31, 26 to 30
and so on upto bits 1 to 5 are:
.1519 .9902 .2728 .2475 .5347 .3797 .3064 .4323 .4766 .9255 .8667
.5735 .3759 .4901 .3992 .2638 .1307 .9250 .6487 .1943 .3605 .2605
.6538 .8069 .0946 .3823 .3940

DNA test for generator ranintb.32 using bits 31 to 32
No. missing words = 1034559, Equiv normal variate = *******,
p-value= 1.0000
p-values obtained by repeating test using bits 30 to 31, 29 to 30
and so on upto bits 1 to 2 are:
1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000
1.0000  .9952  .6402  .9210  .0501
 .5020  .0291  .1165  .8910  .4773
 .5629  .0148  .7519  .6280  .1549
 .2364  .2943  .2364  .9564  .9305

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::     This is the COUNT-THE-1's TEST on a stream of bytes.      ::
:: Consider the file under test as a stream of bytes (four per   ::
:: 32 bit integer).  Each byte can contain from 0 to 8 1's,      ::
:: with probabilities 1,8,28,56,70,56,28,8,1 over 256.  Now let  ::
:: the stream of bytes provide a string of overlapping  5-letter ::
:: words, each "letter" taking values A,B,C,D,E. The letters are ::
:: determined by the number of 1's in a byte::  0,1,or 2 yield A,::
:: 3 yields B, 4 yields C, 5 yields D and 6,7 or 8 yield E. Thus ::
:: we have a monkey at a typewriter hitting five keys with vari- ::
:: ous probabilities (37,56,70,56,37 over 256).  There are 5^5   ::
:: possible 5-letter words, and from a string of 256,000 (over-  ::
:: lapping) 5-letter words, counts are made on the frequencies   ::
:: for each word.   The quadratic form in the weak inverse of    ::
:: the covariance matrix of the cell counts provides a chisquare ::
:: test::  Q5-Q4, the difference of the naive Pearson sums of    ::
:: (OBS-EXP)^2/EXP on counts for 5- and 4-letter cell counts.    ::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
Test results for ranintb.32
Chi-square with 5^5-5^4=2500 d.of f. for sample size:2560000
Results fo COUNT-THE-1's in successive bytes:
                                chisquare  equiv normal  p-value
byte stream for ranintb.32        2485.16      -.210      .416887
byte stream for ranintb.32        2515.64       .221      .587527

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::     This is the COUNT-THE-1's TEST for specific bytes.        ::
:: Consider the file under test as a stream of 32-bit integers.  ::
:: From each integer, a specific byte is chosen , say the left-  ::
:: most::  bits 1 to 8. Each byte can contain from 0 to 8 1's,   ::
:: with probabilitie 1,8,28,56,70,56,28,8,1 over 256.  Now let   ::
:: the specified bytes from successive integers provide a string ::
:: of (overlapping) 5-letter words, each "letter" taking values  ::
```

```
:: A,B,C,D,E. The letters are determined  by the number of 1's,  ::
:: in that byte::  0,1,or 2 ---> A, 3 ---> B, 4 ---> C, 5 ---> D,::
:: and  6,7 or 8 ---> E.  Thus we have a monkey at a typewriter  ::
:: hitting five keys with with various probabilities::  37,56,70,::
:: 56,37 over 256. There are 5^5 possible 5-letter words, and    ::
:: from a string of 256,000 (overlapping) 5-letter words, counts ::
:: are made on the frequencies for each word. The quadratic form ::
:: in the weak inverse of the covariance matrix of the cell      ::
:: counts provides a chisquare test::  Q5-Q4, the difference of  ::
:: the naive Pearson  sums of (OBS-EXP)^2/EXP on counts for 5-   ::
:: and 4-letter cell counts.                                     ::
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
Chi-square with 5^5-5^4=2500 d.of f. for sample size: 256000
For bits  1 to  8, chisquare = 2479.93, equiv normal = -.284,
p-value = .388269
p-values obtained by repeating test using bits 2 to 9, 3 to 10 and
so on upto bits 25 to 32 are:
.573226 .672089 .128618 .576059 .051137 .415617 .191874 .100567
.334250 .747734 .342461 .944903 .199425 .816901 .935848 .863812
.411844 .712764 .508716 .845426 .350002 .735914 .713542 .926592

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::                THIS IS A PARKING LOT TEST                     ::
:: In a square of side 100, randomly "park" a car---a circle of  ::
:: radius 1.   Then try to park a 2nd, a 3rd, and so on, each    ::
:: time parking "by ear".  That is, if an attempt to park a car  ::
:: causes a crash with one already parked, try again at a new    ::
:: random location. (To avoid path problems, consider parking    ::
:: helicopters rather than cars.)   Each attempt leads to either ::
:: a crash or a success, the latter followed by an increment to  ::
:: the list of cars already parked. If we plot n:  the number of ::
:: attempts, versus k::  the number successfully parked, we get a::
:: curve that should be similar to those provided by a perfect   ::
:: random number generator.  Theory for the behavior of such a   ::
:: random curve seems beyond reach, and as graphics displays are ::
:: not available for this battery of tests, a simple characteriz ::
:: ation of the random experiment is used: k, the number of cars ::
:: successfully parked after n=12,000 attempts. Simulation shows ::
:: that k should average 3523 with sigma 21.9 and is very close  ::
:: to normally distributed.  Thus (k-3523)/21.9 should be a st-  ::
:: andard normal variable, which, converted to a uniform varia-  ::
:: ble, provides input to a KSTEST based on a sample of 10.      ::
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
CDPARK: result of ten tests on file ranintb.32
Of 12,000 tries, the average no. of successes should be 3523
with sigma=21.9
test 1 - Successes: 3509    z-score:  -.639 p-value: .261324
p-values obtained by repeating the test 9 more times:
.794438   .607947  .500000
.921543   .481790  .392053
.323972   .572463  .500000
square size = 100, Avg. no. parked = 3525.800,
Sample sigma = 12.584
KSTEST for the above 10 p-vales: p = .571172
```

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::                THE MINIMUM DISTANCE TEST                       ::
:: It does this 100 times::   choose n=8000 random points in a    ::
:: square of side 10000.  Find d, the minimum distance between    ::
:: the (n^2-n)/2 pairs of points.  If the points are truly inde-  ::
:: pendent uniform, then d^2, the square of the minimum distance  ::
:: should be (very close to) exponentially distributed with mean  ::
:: .995 .  Thus 1-exp(-d^2/.995) should be uniform on [0,1) and   ::
:: a KSTEST on the resulting 100 values serves as a test of uni-  ::
:: formity for random points in the square. Test numbers=0 mod 5  ::
:: are printed but the KSTEST is based on the full set of 100     ::
:: random choices of 8000 points in the 10000x10000 square.       ::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
This is the MINIMUM DISTANCE test
for random integers in the file ranintb.32
Sample no.    d^2     avg      equiv uni
      5     2.5587  2.2404     .923582
     10     1.2101  1.5214     .703634
     15      .2147  1.2754     .194066
     20     1.4283  1.1883     .761988
     25      .0912  1.0207     .087594
     30     1.6336  1.0923     .806364
     35     1.2613  1.0981     .718498
     40      .6804  1.1366     .495307
     45      .8529  1.2228     .575638
     50      .1311  1.1856     .123479
     55      .5743  1.1740     .438539
     60      .3058  1.1867     .264567
     65      .3740  1.1796     .313302
     70      .7902  1.1945     .548046
     75     1.1124  1.1619     .673061
     80      .5055  1.1514     .398339
     85      .8237  1.1276     .562990
     90      .9028  1.0971     .596392
     95      .0301  1.0912     .029771
    100      .1171  1.0892     .111026
MINIMUM DISTANCE TEST for ranintb.32
Result of KS test on 20 transformed mindist^2's:
p-value= .848977

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::              THE 3DSPHERES TEST                                ::
:: Choose  4000 random points in a cube of edge 1000.  At each    ::
:: point, center a sphere large enough to reach the next closest  ::
:: point. Then the volume of the smallest such sphere is (very    ::
:: close to) exponentially distributed with mean 120pi/3.  Thus   ::
:: the radius cubed is exponential with mean 30. (The mean is     ::
:: obtained by extensive simulation).  The 3DSPHERES test gener-  ::
:: ates 4000 such spheres 20 times.  Each min radius cubed leads  ::
:: to a uniform variable by means of 1-exp(-r^3/30.), then a      ::

::  KSTEST is done on the 20 p-values.                            ::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
The 3DSPHERES test for file ranintb.32
sample no:  1     r^3=  14.229     p-value= .37768
sample no:  2     r^3=  10.512     p-value= .29559
Test is similarly repeated a total of 20 times.
A KS test is applied to those 20 p-values.
3DSPHERES test for file ranintb.32 p-value= .213839

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::      This is the SQEEZE test                                    ::
::  Random integers are floated to get uniforms on [0,1). Start-   ::
::  ing with k=2^31=2147483647, the test finds j, the number of    ::
::  iterations necessary to reduce k to 1, using the reduction     ::
::  k=ceiling(k*U), with U provided by floating integers from      ::
::  the file being tested.  Such j's are found 100,000 times,      ::
::  then counts for the number of times j was <=6,7,...,47,>=48    ::
::  are used to provide a chi-square test for cell frequencies.    ::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
RESULTS OF SQUEEZE TEST FOR ranintb.32
Table of standardized frequency counts ( (obs-exp)/sqrt(exp) )^2
for j taking values <=6,7,8,...,47,>=48:
-1.5       .9      -.1     -1.3      1.0      1.6
  .1       .9      1.2       .9      1.5       .1
  .2      -.7       .8      -.9       .0      -.2
 -.6      -.2      -.7      -.2       .2      -.1
  .4      -.8      -.8       .2      -.3      1.8
  .3     -1.1       .3       .5      -.2     -1.0
  .0      1.5       .1      -.7      -.6     -1.0
-1.1
Chi-square with 42 degrees of freedom: 28.847
z-score= -1.435  p-value= .061124

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::             The  OVERLAPPING SUMS test                          ::
:: Integers are floated to get a sequence U(1),U(2),... of uni-    ::
:: form [0,1) variables.  Then overlapping sums,                   ::
::   S(1)=U(1)+...+U(100), S2=U(2)+...+U(101),... are formed.      ::
:: The S's are virtually normal with a certain covariance mat-     ::
:: rix.  A linear transformation of the S's converts them to a     ::
:: sequence of independent standard normals, which are converted ::
:: to uniform variables for a KSTEST. The  p-values from ten       ::
:: KSTESTs are given still another KSTEST.                         ::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
p-values for ten tests are :
.896026  .309515  .690461  .233085  .178876
.576975  .629410  .544685  .890147  .486059
Results of the OSUM test for ranintb.32
KSTEST on the above 10 p-values:  .224907

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: This is the CRAPS TEST. It plays 200,000 games of craps, finds::
:: the number of wins and the number of throws necessary to end  ::
:: each game.  The number of wins should be (very close to) a     ::
:: normal with mean 200000p and variance 200000p(1-p), with       ::
:: p=244/495.  Throws necessary to complete the game can vary     ::
:: from 1 to infinity, but counts for all>21 are lumped with 21. ::
:: A chi-square test is made on the no.-of-throws cell counts.   ::
:: Each 32-bit integer from the test file provides the value for ::
```

```
:: the throw of a die, by floating to [0,1), multiplying by 6    ::
:: and taking 1 plus the integer part of the result.             ::
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
Results of craps test for ranintb.32
No. of wins:
Observed = 98637, Expected = 98585.86
z-score = .229, pvalue= .59046
Analysis of Throws-per-Game:
Chisq =  14.09 for 20 degrees of freedom, p=  .17432
Throws Observed Expected   Chisq      Sum
   1     66529    66666.7    .284      .284
   2     37637    37654.3    .008      .292
   3     27144    26954.7   1.329     1.621
   4     19173    19313.5   1.022     2.643
   5     13775    13851.4    .422     3.064
   6     10028     9943.5    .717     3.782
   7      7276     7145.0   2.401     6.183
   8      5217     5139.1   1.182     7.364
   9      3682     3699.9    .086     7.451
  10      2630     2666.3    .494     7.945
  11      1919     1923.3    .010     7.955
  12      1352     1388.7    .972     8.926
  13       984     1003.7    .387     9.314
  14       741      726.1    .304     9.618
  15       547      525.8    .852    10.470
  16       381      381.2    .000    10.470
  17       285      276.5    .259    10.729
  18       193      200.8    .305    11.034
  19       147      146.0    .007    11.041
  20        93      106.2   1.644    12.685
  21       267      287.1   1.409    14.094
 SUMMARY  FOR ranintb.32
 p-value for no. of wins: .590461
 p-value for throws/game: .174323
```

VITA

Mahesh Gopalan was born in Salem, Tamil Nadu, India on August 8, 1976. He attended school at Chinmaya Vidyalaya in Bokaro and Sri Vidya Mandir in Salem, and graduated high school from Petit Seminaire in Pondicherry.

With the aid of a good performance in the Tamil Nadu Professional Colleges Entrance Examination, he was admitted into the Coimbatore Institute of Technology for a Baccalaureate degree in Engineering with a concentration in Electronics and Communication. An active participant in extra-curricular activities, he served as Associate Editor of the college magazine, and as Associate Secretary of the astronomy club of C.I.T. He also benefited with a summer internship position at Salem Steel Plant, Salem. He graduated first class in the spring of 1997.

He continued his education by pursuing a Master of Science degree in Electrical Engineering at the University of Tennessee Space Institute, part of the University of Tennessee at Knoxville. He served as a Graduate Research Assistant during the course of the Masters program and graduated in the summer of 1999. He was chosen by the University as one of its nominees for the 1999 edition of Who's Who Among Students in American Colleges and Universities.