RN

Random Numbers

Remarks

Keywords: Random, pseudorandom, linear congruential, lagged Fibonacci

1. List of prerequisites:
   some exposure to sequences and series, some ability to work in base 2 arithmetic (helpful), good background in FORTRAN or some other procedural language.

2. List of computational methods:
   pseudorandom numbers; random numbers; linear, congruential generators (LCGs); lagged Fibonacci generators (LFGs).

3. List of architectures/computers:
   any computer with a FORTRAN compiler and bitwise logical functions.

4. List of codes supplied:
   `ranlc.f(or)` - linear, congruential generator,
   `ranlf.f(or)` - lagged Fibonacci generator,

   `random.f(or)` - "portable" linear, congruential generator, after Park and Miller, 1988.
   `getseed.f(or)` - to generate initial seeds from the time and date, implemented for Unix and IBM PC's.

5. Scope:
   2-6 lectures, depending upon depth of coverage.

Notation Key

a	integer multiplier, $0 < a \leq m-1$
c	integer constant, $0 \leq c \leq m-1$
D	length of needle in Buffon's needle experiment

F	fraction of trials needle falls within ruled grid in Buffon's needle experiment
k	lag in lagged Fibonacci generator, $k > 0$
ℓ	lag in lagged Fibonacci generator, $\ell > k$
M	number of binary bits in m
m	integer modulus $m > 1$
N	number of trials
N_p	number of parallel processors
P	period of generator
p	a prime integer
R_n	random real number, $0 \leq R_n < 1$
S	spacing between grid in Buffon's needle experiment
X_n	n^{th} random integer

Subscripts and Superscripts

0	denotes initial value, or seed
j	denotes j^{th} value
ℓ	denotes ℓ^{th} value
k	denotes k^{th} value
n	denotes n^{th} value

1 Monte Carlo Algorithms and "Random" Numbers

In this chapter, we discuss the application of Monte Carlo methods to scientific and engineering problems. There are problems which can be solved only by the Monte Carlo method, and there also are problems which are best solved by the Monte Carlo method. However, the method is often referred to as the "method of last resort," as it is apt to consume large computing resources. As such, it is a method ideally suitable for inclusion in a textbook on computational science, as Monte Carlo programs, due to their nature of consuming vast computing resources, have historically had to be executed upon the fastest computers available at the time, and employ the most advanced algorithms, implemented with substantial programming acumen.

Here, we provide a brief history of the Monte Carlo method. A good early work is that of Buffon's needle problem [Ross, 1976], where in 1768 Buffon, a French mathematician, experimentally determined a value of π by casting a needle on a ruled grid. Many trials were required to obtain good accuracy. The objective of the work was a validation of statistical theory, as opposed to Monte Carlo simulations typically applied today, where answers to new physical problems are sought. Lord Rayleigh [Rayleigh, 1899] even delved into this field near the turn of the century. Courant, Fredericks and Levy in 1928 showed how the method could be used to solve boundary value problems. Ensuing work [Hurd, 1949] illustrated that

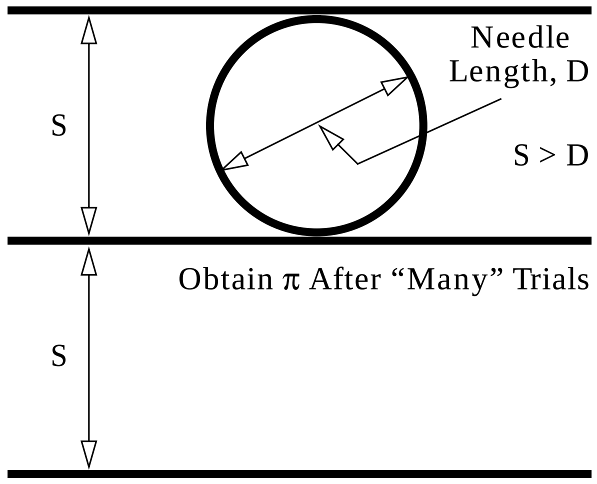

Figure 1: Illustration of Buffon's Needle Problem

if the entire field were sought, Monte Carlo methods are far from competitive. However, if the solution in a restricted region of space is sought (say at a point), then Monte Carlo methods may be used to great advantage, as they may be implemented in restricted regions of space and/or time.

Exercise 1 - Buffon's Needle Problem, to illustrate geometrically some of the properties of random numbers

(a) Write a program to determine the value of π by numerically casting a needle of length D on a ruled grid with spacing S as shown in Figure 1. You must first determine a random position (relative ruled grid as shown in Figure 1. You must first determine a random position (relative to a ruled line), then determine a random angle, and thirdly, see whether the needle falls within the ruled lines. You may find it instructive to compute approximations for π versus the number of trials, e.g. N = 10, 100, 1,000 and 10,000. For $S > D$, your answer for the fraction of time, F, the needle falls within the grid should be [Kalos, 1986]:

$$F = 1 - \frac{2D}{\pi S}$$

(b) Verify the above equation. What is the corresponding formula for $S < D$? (Warning: there is considerable effort involved in this derivation!)

(c) Using a needle and a piece of paper upon which you have placed a ruled grid, perform a physical experiment to determine π using the above equation. Perform first 10, then 100, and then 1,000 trials. Comment on the accuracy of these results relative to the accuracy of the results from the numerical experiment. Discuss reasons for any bias apparent in the results.

Enrico Fermi in the 1930's used Monte Carlo in the calculation of neutron diffusion, and later designed the Fermiac, a Monte Carlo mechanical device used in the calculation of criticality in nuclear reactors. In the 1940's, a formal foundation for the Monte Carlo method was developed by von Neumann, who established the mathematical basis for probability density functions (PDFs), inverse cumulative distribution functions (CDFs), and pseudo-random number generators. The work was done in collaboration with Stanislaw Ulam, who realized the importance of the digital computer in the implementation of the approach. The collaboration resulted from work on the Manhattan project, where the ENIAC was employed in the calculation of yield [Ulam et al., 1947; Ulam and Metropolis, 1949; Eckhard, 1987; Metropolis, 1987].

Individuals in the IBM corporation were pioneers in the field of random number generation, perhaps because they were first engaged in it due to their participation in the Manhattan project, where Richard Feynman then directed their computing operations (a fascinating exposition of their approach to performing large-scale computing involving a parallel approach exists in Richard Feynman's *Surely You're Joking, Mr. Feynman*). It is interesting to note the extremely primitive computing environments in existence at that time, and the challenges this presented to researchers (see Hurd, 1949).

Uses of Monte Carlo methods have been many and varied since that time. However, due to computer limitations, the method has not yet fully lived up to its potential as discussed by Metropolis [Metropolis, 1985]. Indeed, this is reflected in the stages the method has undergone in the fields of engineering. In the late 1950's and 1960's, the method was tested in a variety of engineering fields [Meuller, 1956; Ehrlich, 1959; Polgar and Howell, 1965; Haji-Sheikh, 1968 and Chandler et al., 1968]. At that time, even simple problems were compute-bound. The method has since been extended to more complex problems [Howell, 1968; Modest, 1978; Maltby, 1987a and 1987b, Maltby and Burns, 1988 and 1989; Crockett et al., 1989]. Since then, attention was focused upon much-needed convergence enhancement procedures [Kahn and Marshall, 1953; Emery and Carson, 1968; Burghart and Stevens, 1971; Lanore, 1971; Shamsunder et al., 1973; Zinsmeister and Sawyer, 1976; Larsen and Howell, 1986]. Many complex problems remained intractable through the seventies.

With the advent of high-speed supercomputers, the field has received increased attention, particularly with parallel algorithms which have much higher execution rates. In his Ph.D. dissertation, Brown introduced the concept of the "event step" [Brown, 1981], enabling efficient vectorization of Monte Carlo algorithms where the particles do not interact. This approach was later successfully exploited by several investigators. Martin et al. [Martin et al., 1986] reported speedups of a factor of five on an IBM 3090 with vector units. Nearly linear speedup was reported [Sequent Computer Systems, 1985] on a parallel architecture for photon tracing. Bobrowicz et al. [Bobrowicz et al., 1984a and 1984b] obtained speedup factors of from five to eight in an algorithm where particles are accumulated in queues until efficient vector lengths are obtained, allowing physics algorithms such as the Los Alamos benchmark GAMTEB to be effectively vectorized [Burns et al., 1988]. Such advanced coding techniques have enabled much bigger problems to be attacked, with improved accuracy. However, there are still a host of problems which remain intractable, even with an effectively vectorized algorithm.

Moreover, some impediments to effective vectorization have been identified and analyzed. Zhong and Kalos [Zhong and Kalos, 1983] analyzed the "straggler" problem, where few particles persist for many event steps, inhibiting performance due to the overhead incurred where vectors are "short." Pryor and Burns [Pryor and Burns, 1988; Burns and Pryor, 1989] reported, for one Monte Carlo problem where particles interact, speedups on the order of half of those observed where the particles do not interact, albeit with a vectorized algorithm of greatly increased complexity. The same problem has been attacked with different physics [McDonald and Baganoff, 1988; Dagum, 1989] in a more efficient algorithm.

Good general references on Monte Carlo abound [Beckenback, 1956; Hammersly and Handscomb, 1964; Schreider, 1964; Kleinjnen, 1974; Rubenstein, 1981; Binder, 1984; Haji-Sheikh, 1988] in the literature. Most have a distinct bent - usually either statistics or physics. Unfortunately, there is a dearth involving large-scale engineering applications.

No tutorial on large-scale Monte Carlo simulation can be complete without a discussion of pseudo-random number generators. Where billions of random numbers are required, it is essential that the generator be of long period, have good statistical properties and be vectorizable. On 64-bit machines, these criteria are usually satisfied with a multiplicative generator if the constants are carefully chosen [Kalos and Whitlock, 1986]. Indeed, we have come a long way from the early days of random number generators [Knuth, 1981]. For example, Cray's FORTRAN callable generator `ranf` has a cycle length of 2^{44}, and is very efficient (vectorized with low overhead).

We summarize this introduction with our impression of the present status of Monte Carlo surface to surface simulation. "All the pieces of the puzzle" have just come into confluence for large-scale Monte Carlo analysis. First, supercomputers are now sufficiently powerful to enable the simulation of very large engineering and physics systems, involving thousands of surfaces and billions of particle emissions. Secondly, a comprehensive formulation for material properties exists in the aggregate of several models. Thirdly, an estimate of the number of trials required to achieve a specified level of accuracy is now obtainable prior to execution. This makes possible a formulation which allows the number of emissions to evolve dynamically as the simulation proceeds. Finally, a number of investigators have effectively vectorized diverse Monte Carlo transport algorithms, with a sufficient base to establish a synthesized approach. We now even have a quantitative model which allows the assessment of the degree of parallelism and the amount of overhead required. Moreover, with the emergence of lagged Fibonacci generators, parallelization at any granularity appears to be easily implemented, and robust. With this as a point of departure, we now embark upon a discussion of random number generators, upon which all Monte Carlo methods rely.

2 Introduction to Pseudorandom Numbers

Anyone who considers arithmetical methods of producing random digits is, of *course, in a state of sin.* – John von Neumann (1951)

Anyone who has not seen the above quotation in at least 100 places is probably not very old. – D. V. Pryor (1993)

In this section, we shall explore possible reasons for von Neumann's curious statement. This statement is curious for two particular reasons: (1) arithmetic methods are widely used on all computers to generate random numbers, and (2) John von Neumann himself devised such methods for generating random numbers. Von Neumann probably made the above statement in reference to the fact that there are many ways to go amiss when so doing. Our goal in this section is, through a series of example problems, to illustrate various properties of random number generators. With a visceral understanding of the generation process and a solid foundation of cause and effect illustrated through examples, it is our hope that this may help prevent "straying" into unacceptable techniques.

Indeed, all random number generators are based upon specific mathematical algorithms, which are repeatable and sequential. As such, the numbers are just *pseudo*random. Here, for simplicity, we shall term them just "random" numbers, subject to this realization. Formally,

Truly random - is defined as exhibiting "true" randomness, such as the time between "tics" from a Geiger counter exposed to a radioactive element.

Pseudorandom - is defined as having the *appearance* of randomness, but nevertheless exhibiting a specific, repeatable pattern.

Quasi-random - is defined as filling the solution space sequentially (in fact, these sequences are not at all random - they are just comprehensive at a preset level of granularity). For example, consider the integer space [0, 100]. One quasi-random sequence which fills that space is 0, 1, 2,...,99, 100. Another is 100, 99, 98,...,2, 1, 0. Yet a third is 23, 24, 25,..., 99, 100, 0, 1,..., 21, 22. Pseudorandom sequences which would fill the space are pseudorandom permutations of this set (they contain the same numbers, but in a different, "random" order).

As Monte Carlo simulations have been developed on computers since their inception, methods of generating and dealing with random numbers are fairly well established. As literally millions or even billions of random numbers are required in a large-scale simulation, the process should ideally be very efficient. Indeed, many random number generator routines are written in assembly language for this reason. Sometimes, random number routines are even put "in line" to avoid the overhead associated with subroutine calls. Often on vector and/or parallel computers, blocks of random numbers are generated to amortize, over many random numbers, the overhead associated with the generation of one random number. It is noteworthy that the FORTRAN 90 ANSI standard is the first one to identify a random number generator in the ANSI specification of the language! Hitherto, it has existed only as an extension to the FORTRAN standard.

We shall not provide a comprehensive treatise on random numbers. There are many good references which already do so. Knuth [Knuth, 1981] is the definitive reference, although it may be a bit too abstract for the typical computational scientist. Some implementations of algorithms (of which, at large scale, some are good and some are not so good) can be found in *Numerical Recipes* [Press et al., 1990]. Perhaps the most accessible and lucid exposition is that of Anderson [Anderson, 1990], who presents some excellent illustrative graphics - an approach which we emulate to illustrate the properties of some random number generators. Also, our focus is upon efficiency and effectiveness when using random number generators in large-scale computations.

> *Random number generators should not be chosen at random.* – Donald Knuth (1986)

Consistent with the previous citation, we advise a modicum of caution in the use of pseudorandom numbers - especially in large-scale problems. There is an interesting anecdote from Knuth, who went to great lengths to implement what he thought was to be a superior random number generator. However, upon testing, it was found to produce very poor random numbers, illustrating that it is easy for even the experts *a priori* to misinterpret quality. The following comments derive from painful personal experiences of one of the authors.

- When problems arise with random number generators, they are exceedingly difficult to isolate. Often, the problem can be isolated only by replacing the existing random number generator with one which is definitely superior (although perhaps much slower). To make a selection of a superior generator requires both knowledge of the generator currently in use and sometimes requires in-depth knowledge of the properties of general random number generators.

- Problems rarely occur when solving small test problems (those for which analytical or experimental answers are known). Instead, problems arise in large scale, substantial examples involving perhaps millions or even billions of random numbers - where debugging is difficult due to the massive amount of data.

- Finally, in large-scale problems where one is porting a scalar to a vector or massively parallel algorithm, the random numbers usually are accessed in a different order. Thus it is sometimes not possible to duplicate the run exactly. The results converge only asymptotically, as the number of trials increases.

Thus, when problems occur, it is very difficult to isolate the problem to the random number generator because one tends to trace program execution an event step at a time, and it is only in aggregate over many random numbers that the behavior of the random number generator is flawed. In effect, one "loses sight of the forest for all of the trees." Typically, in desperation and as a last resort after many days of debugging, one changes the random number generator and voila - the problem disappears!

The truly cautious researcher assesses different random number generators as the continuum analyst makes refinements to a grid — better and better random number generators are employed, until the answers are independent of the random number generator. This is rarely, if ever, done in practice. Waxing philosophical, one wonders what number of Monte Carlo simulations may have been performed where the answers may in fact be incorrect, but not grossly incorrect, due to a flaw inherent in the random number generator used. Traditionally, we cavalierly accept the random number generator on the architecture of interest. Fortunately, due to the early and well publicized mistakes made using random number generators, their properties were thoroughly investigated by the mathematics community, primarily in the 1950's. Most of the random number generators in use today were designed with cognizance of past pitfalls and are adequate for almost all applications. However, we conclude this section with a firm *caveat emptor!*

2.1 Desirable Properties

When performing Monte Carlo simulation, we use random numbers to determine: (1) attributes (such as outgoing direction, energy, etc.) for launched particles, and (2) interactions of particles with the medium. Viewing this process physically, the following properties are desirable:

- The attributes of particles should not be correlated. That is, the attributes of each particle should be independent of those attributes of any other particle.

- The attributes of particles should be able to fill the entire attribute space in a manner which is consistent with the physics. For example, if we are launching particles into a hemispherical space above a surface, then we should be able to approach completely filling the hemisphere with outgoing directions, as we approach an infinite number of particles launched. At the very least, "holes" or sparseness in the outgoing directions should not affect the answers significantly. Also, if we are sampling from an energy distribution, with an increasing number of particles, we should be able to duplicate the energy distribution better and better, until our simulated distribution is "good enough."

Mathematically speaking, the sequence of random numbers used to effect a Monte Carlo model should possess the following properties:

Uncorrelated Sequences - The sequences of random numbers should be serially *uncorrelated.* This means that any subsequence of random numbers should not be correlated with any other subsequence of random numbers. Most especially, n-tuples of random numbers should be independent of one another. For example, if we are using the random number generator to generate outgoing directions so as to fill the hemispherical space above a point (or area), we should generate no unacceptable geometrical patterns in the distribution of outgoing directions.

Long Period - The generator should be of *long period* (ideally, the generator should not repeat; practically, the repetition should occur only after the generation of a very large set of random numbers). More explanation is provided below.

Uniformity - The sequence of random numbers should be uniform, and unbiased. That is, equal fractions of random numbers should fall into equal "areas" in space. For example, if random numbers on [0,1) are to be generated, it would be poor practice were more than half to fall into [0, 0.1), presuming the sample size is sufficiently large. Often, when there is a lack of uniformity, there are n-tuples of random numbers which are correlated. In this case, the space might be filled in a definite, easily observable pattern. Thus, the properties of uniformity and uncorrelated sequences are loosely related.

Efficiency - The generator should be efficient. In particular, the generator used on vector machines should be vectorizable, with low overhead. On massively parallel architectures, the processors should not have to communicate among themselves, except perhaps during initialization. This is not generally a significant issue. With minimal effort, random number generators can be implemented in a high level language such as C or FORTRAN, and be observed to consume well less than 1% of overall CPU time over a large suite of applications.

Exercise 2 - To illustrate the mean and variance of uniformly distributed random numbers

For uniformly distributed real random numbers, R_n, on [0,1), the average value should be 0.5. Note that, to compute the mean and variance, we can order the random numbers in any fashion (as both of these computations are independent of order). It is useful to choose to order the real, random numbers in ascending order, and approximate the distribution in the form of a continuous variable R. Verify by direct integration that the average, $\bar{R}$, and the variance, σ^2, should be 0.5 and 0.083333, respectively, as follows:

$$\bar{R} = \int_0^1 R dR = \frac{1}{2}$$

$$\sigma^2 = \int_0^1 (R - 0.5)^2 dR = \frac{1}{12}$$

2.2 The Random Number Cycle

Pursuant to the above discussion, it is useful to present pseudorandom sequences in the context of their cyclical structure. Almost all random number generators have as their basis a sequence of pseudorandom integers (there are exceptions). The integers or "fixed point" numbers are manipulated arithmetically to yield floating point or "real" numbers. The

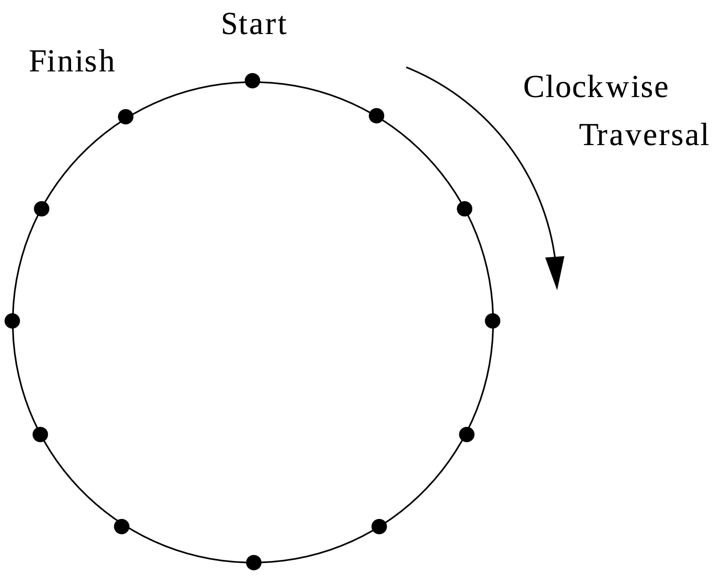

Figure 2: Illustration of Random Number Cycle

random number cycle can be presented in terms of either integers or real numbers. Here, for clarity, we confine ourselves to integers.

In Figure 2, we illustrate a random number cycle representing a sequence of 12 integers. Each black dot represents a distinct integer. Our convention will be to start at the top of the cycle (i.e. 12 o'clock), traverse the cycle clockwise, and finish at the integer just to the left of the start. The nature of the cycle illustrates that: (1) the sequence has a finite number of integers, (2) the sequence gets traversed in a particular order, and (3) the sequence repeats if the period of the generator is exceeded (i.e., the cycle can be traversed more than once). Furthermore, the integers need not be distinct; that is, they may repeat. We shall address this point subsequently. These all are properties of pseudorandom sequences of integers. In subsequent sections, we shall illustrate these aspects with specific examples.

3 Linear, Congruential Generators

3.1 Approach

We begin by discussing the linear congruential generator - the one most commonly used for generating random integers.

$$X_{n+1} = aX_n + c \pmod{m}$$

Here, we generate the next random integer X_{n+1} using the previous random integer X_n, the integer constants a and c, and the integer modulus m. After the integer $aX_n + c$ is generated, modulo arithmetic using the modulus m is performed, to yield the new "random" integer X_{n+1}.

To get started, the algorithm requires an initial "seed," X_0, which must be provided by some means (we shall discuss this later). The entire sequence is characterized by the multiplier, a; the additive constant, c; the modulus, m; and the initial seed X_0. Following Anderson [Anderson, 1990], we therefore refer to the sequence generated as $LCG(a, c, m, X_0)$, which completely determines the sequence. Here, LCG denotes a Linear, Congruential Generator.

The appearance of randomness is provided by performing modulo arithmetic or remaindering. For example, the nonnegative integers $0, 1, 2, 3, 4, 5, \ldots$ modulo 3 are $0, 1, 2, 0, 1, 2, \ldots$. Note that the next result, X_{n+1}, depends upon only the previous integer, X_n. This is a characteristic of linear, congruential generators which minimizes storage requirements, but at the same time, imposes restrictions on the period.

With X_n determined, we generate a corresponding real number as follows:

$$R_n = \frac{X_n}{\text{float}(m)} \quad \text{or} \quad R_n = \frac{X_n}{\text{float}(m-1)}$$

When dividing by m, the R_n values are then distributed on [0,1). If we desire R_n to be distributed on [0, 1], then we would divide by $(m - 1)$. We desire uniformity, where any particular R_n is just as likely to appear as any other R_n, and the average of the R_n is very close to 0.5.

Example 1 LCG (5, 1, 16, 1)

Let us consider a simple example with $a = 5$, $c = 1$, $m = 16$, and $X_0 = 1$. The sequence of pseudorandom integers generated by this algorithm is:

1,6,15,12,13,2,11,8,9,14,7,4,5,10,3,0,1,6,15,12,13,2,11,8,9,14,...

In Figure 3, we illustrate the random number cycle for this generator. We immediately observe four features:

- The period (the number of integers before the sequence repeats) P is 16 - exactly equal to the modulus, m. When the next result depends upon only the previous integer, the longest period possible is $P = m$. In the current example with a modulus of 16, the mod operation generates integer results from 0 to 15, inclusive. Thus, for $m = 16$, this sequence is of long period (the longest possible), and uniform (it completely fills the space of integers from 0-15). Note that the period is exactly equal to 2^4, i.e. 2^M, where M is the base 2 log of the modulus.

- This particular sequence exhibits throughout its period the pattern of alternating odd and even integers. It is frequently instructive to view this sequence in binary and then after performing a real division by m to result in random real numbers (see Table 1). It is readily apparent that the sequence is serially correlated. In fact, note that the right-most binary digit exhibits the regular pattern $1, 0, 1, 0, \ldots$. Due to this lack of

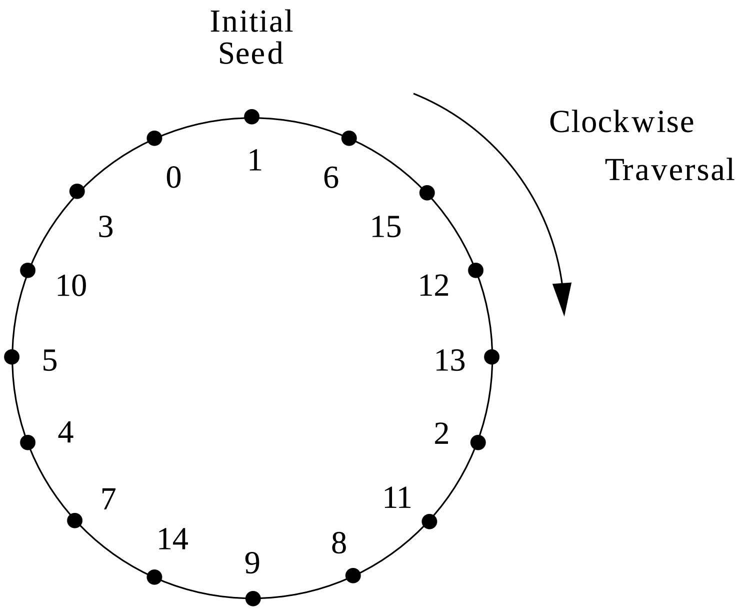

Figure 3: Random Number Cycle for Example 1- LC(5,1,16,1)

randomness, the X_n values should not be used as random digits (especially the right-most digits of X_n). In fact, this lack of randomness results from using a power of two for m. Despite this, a modulus which is a power of two is often used, for this renders the process of performing the modulus operation very efficient. Moreover, the real numbers generated from the integer sequence are generally sufficiently random in the higher order (most significant) bits to be used in many application codes.

- Next, we infer the following. Because each random integer results from the previous integer alone, selecting any initial seed from 0 to 15 would just cyclically shift the above sequence. We could begin anywhere on the random number cycle, and we would proceed clockwise around the random number cycle from that starting point. Thus, all that a different choice of the initial seed does is shift the starting point in the sequence already determined by a, c and m.

- Finally, we note that the average of the real numbers is 0.4688 and the variance is 0.0830. The departure of these values from the ideal ones of $1/2$ and $1/12 \approx 0.08333$ is due to the short period of this sequence and the rather coarse resolution of the generated real numbers. These conditions of average and variance approaching the theoretical values are necessary but not sufficient conditions for a good random number generator.

Let us exercise the code `ranlc.f` for additional parameters. In so doing, we shall choose parameters that illustrate specific aspects of linear, congruential generators. First, we take the case of $c = 0$. This is termed a *multiplicative congruential* random number generator:

Table 1: Random Sequence of Example 1 - $LCG(5,1,16,1)$

n	"Random" Integer - X_n	"Random" Binary - X_n	"Random" Real - X_n
0	1	0001	0.0625
1	6	0110	0.3750
2	15	1111	0.9375
3	12	1100	0.7500
4	13	1101	0.8125
5	2	0010	0.1250
6	11	1011	0.6875
7	8	1000	0.5000
8	9	1001	0.5625
9	14	1110	0.8750
10	7	0111	0.4375
11	4	0100	0.2500
12	5	0101	0.3125
13	10	1010	0.6250
14	3	0011	0.1875
15	0	0000	0.0000
Average			0.4688
Variance			0.0830

$$X_{n+1} = aX_n \pmod{m} \tag{1}$$

The case $c \neq 0$ is termed a *mixed congruential* random number generator.

Example 2 LCG (5, 0, 16, 1)

As illustrated in Figure 4, we obtain the sequence

1,5,9,13,1,5,9,13,...

- Note that we now have a period, P, of only 4 — this is 1/4 the modulus. In fact, when m is a power of 2 (here, $2^M = 2^4$) and $c = 0$, the maximum period is 2^{M-2}. Here again, we note that the low order bits are not random. In fact, the two least significant bits

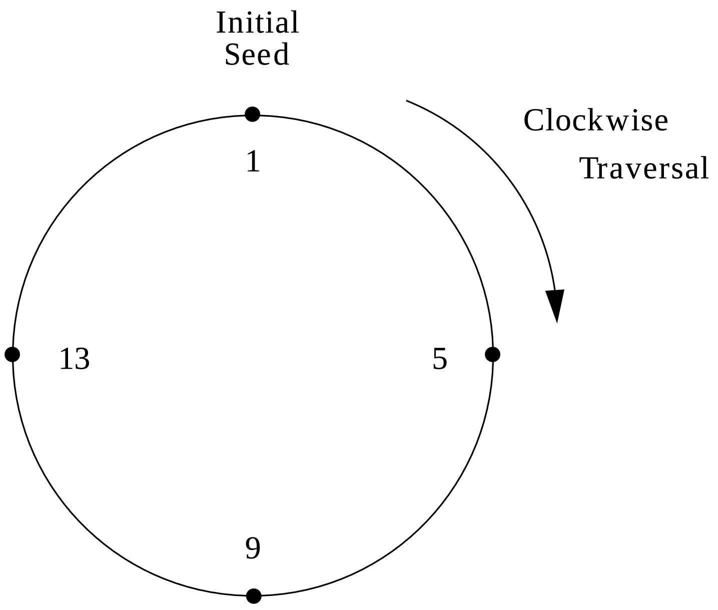

Figure 4: Random Number Cycle for Example 2– LCG(5,0,16,1)

are constant, always 01; and the most significant bits are quasi-random, exhibiting the pattern 00, 01, 10, and 11

- Also, the sequence is correlated, as all successive integers differ by 4 from their predecessors.

- At coarse granularity, the sequence is uniform. For example, if we divide [0,1) equally into quarter segments, then exactly one random number falls into each segment: [0, 0.25), [0.25, 0.5), [0.5, 0.75) and [0.75,1). However, at finer granularity, this uniformity breaks down - consider dividing up the domain into 8 equal segments, for example. Here, we have a mismatch between the random number generator we are using, and the "scale" or granularity of our problem — the RNG is incapable of producing random numbers which fill our attribute space (here, our 8 bins). There are two separate issues to consider here. The least troublesome is the finite precision existing in all computers, which results in a round-off error to the precision with which integers can be represented, or with which the floating point divide is accomplished. More serious is the second issue: the interaction of the sequence of random numbers produced by our generator with our application. This interaction is particularly troublesome when our application requires n-tuples of random numbers, instead of just one random number at a time. Later, we shall illustrate this effect by displaying plots of individual points generated using various RNGs. In some applications, several random numbers are needed at a time. For example, when tracing particles among surfaces, three random numbers are required to obtain outgoing direction (one for outgoing azimuthal angle, and one for outgoing polar angle) and energy (usually, only one is required to obtain

Table 2: Random Sequence of Example 2 - $LCG(5, 0, 16, 1)$

n	"Random" Integer - X_n	"Random" Binary - X_n	"Random" Real - X_n
0	1	0001	0.0625
1	5	0101	0.3125
2	9	1001	0.5625
3	13	1101	0.8125
Average			0.4375
Variance			0.0781

the energy).

Example 3 LCG (5, 0, 37, 1)

We obtain the sequence

1,5,25,14,33,17,11,18,16,6,30,2,10,13,28,29,34,22,
36,32,12,23,4,20,26,19,21,31,7,35,27,24,9,8,3,15, ...

Table 3 provides the sequence throughout the period. Here, because we use a prime number as the divisor for the modulus operation and $c = 0$, we obtain a period one less than modulus 37 (0 is not possible, as it maps to itself, so we obtain a period of 36). Indeed, when $m = p$, a prime, the maximum period, P_{max}, is $m - 1$, even if $c \neq 0$. Thus, for linear, congruential generators with a prime modulus, using a non-zero c does not increase the period.

Here, the low order bits, while not exhibiting a discernible pattern, do not appear as "random" as one might expect. Indeed, as is shown in Altman [Altman, 1988], the bitwise randomnes properties of LCGs should be considered on a case by case basis. He provides examples of LCGs with prime moduli that fail bitwise testing, but points out, for example, that $LCG(13445, 0, 2^{31} - 1, X_0)$ does pass the bitwise randomness test.

Exercise 3 - to illustrate the generation of random numbers using LCG(13445, 0, 65536, 16811) and to partition the random real numbers into bins

Obtain and edit the linear congruential program `ranlc.f`. Supply the additional code to compute the average and the variance. For sample sizes of 10, 100, 1,000, and 10,000, run the code and plot the errors in the average and variance versus sample size. Discuss how the error in these quantities varies with the sample size. Use the following parameters: $a = 16807$, $c = 0$, and $m = 2^{16}(= 65536)$.

3.2 Initial Seed

Now, we address establishing the initial seed. When debugging, it is important to implement the algorithm to reproduce the same stream of random numbers on successive runs. If the run is a debug run (noted, perhaps, by a parameter in the input file), the seed should be set to a constant initial value, such as a large prime number (it should be odd, as this will satisfy period conditions for any modulus). Otherwise, the initial seed should be set to a "random" odd value. Anderson [1990] recommends setting the initial seed X_0 to the following integer:

$$X_0 = iyr + 100 * (imonth - 1 + 12 * (iday - 1 + 31 * (ihour + 24 * (imin + 60 * isec))))$$

where the variables on the right-hand side are the integer values of the date and time. Note that the year is 2 digits long, i.e. the domain of iyr is $[0, 99]$. However, we have found it preferable to introduce the maximum variation in the seed into the least significant bits by using the second of this century, rather than the most significant bits. Ergo, we prefer the following:

$$X_0 = isec + 60 * (imin + 60 * (ihr + 24 * (iday - 1 + 31 * (imon - 1 + 12 * iyr))))$$

and, to ensure X_0 is odd (in FORTRAN):

$$\texttt{x0} = \texttt{ior(x0, 1)}$$

Arithmetical Considerations

Overflow and Negative Integers

Consider performing the operation $aX_n + c \pmod{m}$. A large value of a is desirable to provide sufficient randomness. A large value of m is also desired, so that the period is kept long. For example, on 32 bit computers, a and m are often 31 bits long, as the most significant bit (the 32nd bit) is generally used to indicate sign. When two 31 bit integers are multiplied together, a possibly 62 bit integer results. Thus, an overflow almost always occurs. Fortunately, floating point multipliers (and software emulations thereof) are designed to throw away the most significant bits, and retain the least significant 32 bits.

However, if the result of the multiplication is so as to have the most significant bit (the 32nd bit) set, then the computer may treat this as a negative integer, which is incompatible with the algorithm above. If this happens (it occurs during one-half of the multiply operations), this negative bit must be handled. One strategy to overcome this is to use a bit mask to mask off the most significant bit. In this case, a logical AND operation should be performed on the random, negative integer and the bit mask, `imask=z'7fffffff'`, which is a leading 0 followed by 31 ones (i.e. $01111111111111111111111111111111_2$). This has the effect of the zeroing out the sign bit, forcing the number to be positive. Viz.,

Table 3: Random Sequence of Example 3 - $LCG(5, 0, 37, 1)$

	“Random”	“Random”	“Random”
n	Integer - X_n	Binary - X_n	Real - X_n
0	1	000001	0.0270
1	5	000101	0.1351
2	25	011001	0.6757
3	14	001110	0.3784
4	33	100001	0.8919
5	17	010001	0.4595
6	11	001011	0.2973
7	18	010010	0.4865
8	16	010000	0.4324
9	6	000110	0.1622
10	30	011110	0.8108
11	2	000010	0.0541
12	10	001010	0.2703
13	13	001101	0.3514
14	28	011100	0.7568
15	29	011101	0.7838
16	34	100010	0.9189
17	22	010110	0.5946
18	36	100100	0.9730
19	32	100000	0.8649
20	12	001100	0.3243
21	23	010111	0.6216
22	4	000100	0.1081
23	20	010100	0.5405
24	26	011010	0.7027
25	19	010011	0.5135
26	21	010101	0.5676
27	31	011111	0.8378
28	7	000111	0.1892
29	35	100011	0.9459
30	27	011011	0.7297
31	24	011000	0.6486
32	9	001001	0.2432

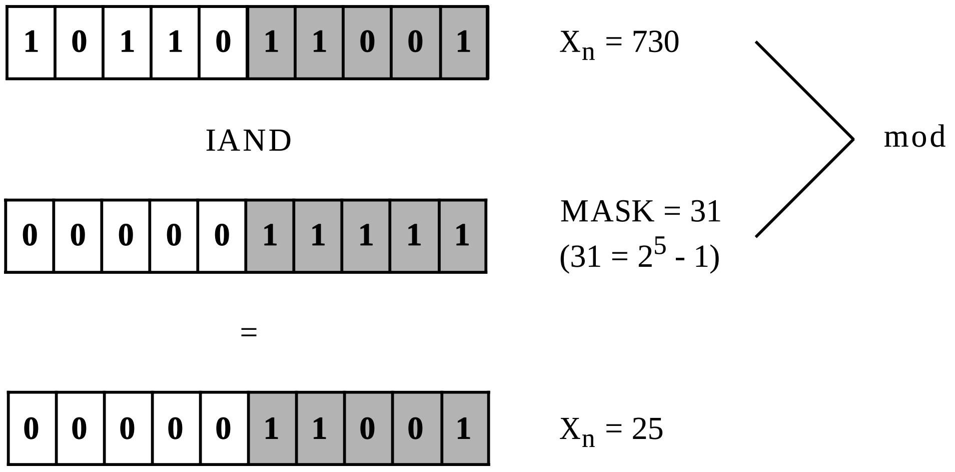

Figure 5: Modulus Operation with $m = 2^5$ using IAND

```
xn = iand(xn, imask)
```

However, masking works only when m is a power of two. Otherwise, it destroys the sequence of numbers generated, and the theory no longer applies. In the general case, unless the number resulting from the multiplication aX_n fits in the register, special coding must be done. An approach to this is to perform remaindering, by decomposing the multiply operation into steps in which intermediate results are contained in 32 bits. This is the strategy used by Park and Miller [Park and Miller, 1988]. Note that, in the language C, one can avoid the issue of the sign bit by using an unsigned integer.

3.3 Using Logical Masks to Perform Modulo 2^M Operations

Generators where the modulus operation must be performed (requiring an integer division) are more costly to implement than are those with moduli of 2^M, where the integer division and remaindering can be accomplished much more efficiently. We illustrate this as follows. With a divisor of 2^M, after the multiplication of aX_n, the next seed is obtained simply by performing a logical AND of X_n with a mask of $(M - 1)$ ones, right-justified (see Figure 5). This is an extremely efficient operation on binary computers. Many computer languages have a bitwise AND intrinsic function. Use of the AND operation also avoids the problem with negative integers, discussed above. A FORTRAN implementation would look like:

```
xn = iand(xn, mask)
```

3.4 Summary of LCG Properties

Following Anderson [Anderson, 1990], we summarize the salient features and the recommendations for three widely used types of linear, congruential random number generators.

Multiplicative, congruential generators are adequate to good for many applications. They are not acceptable... for high-dimensional work. They can be very good if speed is a major consideration. Prime moduli are best. However, moduli of the form 2^N are faster on binary computers. – Anderson (1990)

$m = 2^M, c > 0$:

The full period of m = 2^M is obtained if and only if $a \equiv 1 \pmod 4$, and c is odd (often chosen as 1). Low-order bits are not random!

$m = 2^M, c = 0$:

The maximum period of this generator is 2^{M-2} (one-quarter the modulus), and is obtained if and only if $a \equiv 3 \pmod 8$ or $a \equiv 5 \pmod 8$ (5 is preferred) and the initial seed is odd. Low-order bits are not random!

$m = p$ (prime), $c = 0$, or $c \neq 0$:

The maximum period of this generator is $p - 1$ and is obtained if and only if a is a primitive element modulo p. Note that there is always one integer which maps only to itself. (In the case of $c = 0$, $X_0 = 0$ maps to itself.) Low-order bits may or may not be random! Park and Miller [Park and Miller, 1988] recommend the following, portable generator: $m = p = 2^{31} - 1$ $(2,147,483,647)$, $a = 16,807$, and $c = 0$. The code `random.f`, based upon Park and Miller's algorithm, is supplied with this package. Note that, even when c is non-zero, the maximum period of this generator is still one less than the modulus.

Exercise 4 - to establish all cycles in LCGs when $m = p$, a prime

Consider linear, congruential random number generators. If $c = 0$, it is obvious that $X_0 = 0$ is not a good candidate integer for the initial seed because it maps to itself. In fact, if $m = p$, a prime, then there is always a number which maps to itself (a constant sequence), even if $c \neq 0$. Prove this by finding the integer which maps to itself, and which does not appear in the full period sequence of length $m - 1$ for the following linear, congruential generators:

(a) $LCG(5, 0, 37, X_0)$
(b) $LCG(5, 1, 37, X_0)$
(c) $LCG(5, 5, 37, X_0)$

Exercise 5 - exploring all possible sequences in simple LCGs

For the linear, congruential generator $LCG(5, 0, 16, X_0)$, generate all the possible sequences by varying the initial seed from 0 and 15. How many independent (i.e. not cyclical shifts of one another) sequences are there? Is there a pattern observable in even versus odd integers? Discuss possible reasons for this behavior.

Exercise 6 - Cray's LCG `ranf()`

On a Cray supercomputer running UNICOS, obtain the manual page on its intrinsic random number generator by typing at the UNICOS prompt (%) "`man ranf > ranf.doc`". Read the file `ranf.doc` to see how to set and get the seed for `ranf`. Given that `ranf` is a multiplicative, congruential generator (c = 0) using 46 bits - get the initial seed; discuss it in terms of cycle length. Set the seed to 1, and call `ranf` to obtain a new seed. Get and print out the new seed; this new seed is the multiplier of the generator. Discuss the multiplier in the context of the cycle length. Print out the first ten random integers in decimal and hexadecimal format, and the corresponding random real numbers using an f20.16 format.

4 N-tuple Generation With LCGs

Suppose we wanted to choose random locations in the unit square for Monte Carlo trials in an application program. A very easy way to choose these points is to select two random values, R_1 and R_2, in [0,1) and choosing the point (R_1, R_2) as the current point of interest in the square. More generally, we would generate a point by plotting R_{n+1} vs. R_n, where R_n and R_{n+1} are of course obtained by scaling successive outputs, X_n and X_{n+1}, of the generator $LCG(a, c, m, X_0)$ by $1/m$. If we repeat this procedure for a large number of trials, we would like to expect that we will achieve a reasonably good "covering" of the unit square and in a "randomly" ordered fashion. We would be suspicious of a generator that produced points in the square that were, say, clustered in the bottom half, or perhaps covered the square in some clear order from left to right. Many other forms of obvious nonrandom behavior would be equally unacceptable for most Monte Carlo applications. A characteristic of LCGs is that points selected in this way and plotted in the unit square begin to form regular-looking rows or dotted lines that are easily discernible when enough points have been plotted and when viewed at the proper scale. Over the entire period of an LCG, if all consecutive pairs are plotted, then these rows fill in to become evenly spaced between points.

This constitutes one of the well known tests of randomness applied to pseudo-random number generators - the so-called spectral test, the object of which is to discover the behavior of a generator when its ouputs are used to form n-tuples. The formal development of this family of tests is difficult and we will not cover it here (the interested reader is referred to Knuth), but we will attempt to illustrate through several examples the meaning of the concept as it applies to a 2-dimensional setting and with linear congruential generators. The concepts presented are valid in higher dimensions as well.

In Figure 6a-f we present a few examples to illustrate this behavior. These examples depict the result of generating all pairs of consecutive numbers in the period of full-period LCGs. For each modulus, the plotted points show the effect of our choice of the multiplier, a. In Figure 6a we show the set of points produced by scaling by 1/509 the output of the generator $LCG(10, 0, 509, 1)$. Note that the points form more than one set of rows. That is, there are several angles from which the points appear to line up. From some perspectives, the rows are close together, while from at least one perspective, the rows are rather far apart.

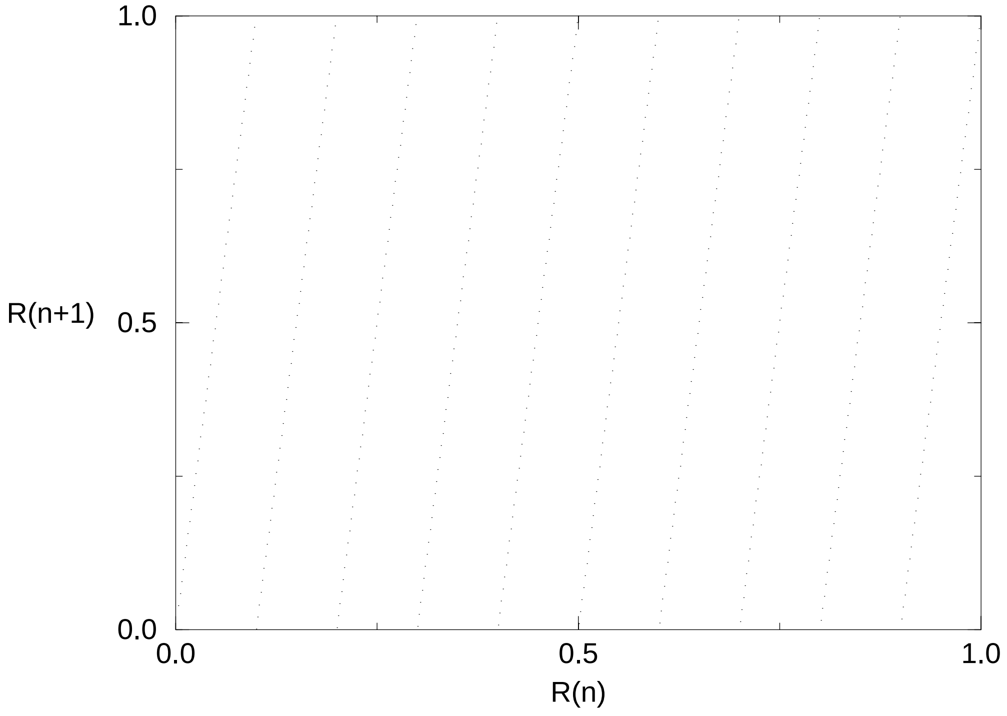

Figure 6: (a) Pair Plot of LCG(10,0,509,1)

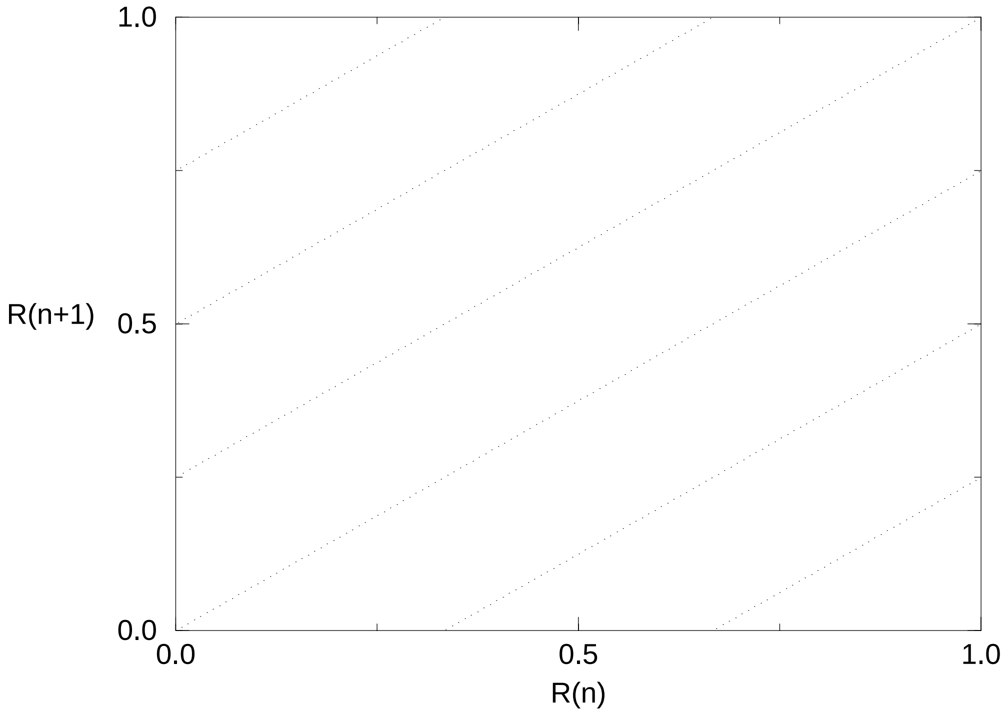

Figure 6: (b) Pair Plot of LCG(128,0,509,1)

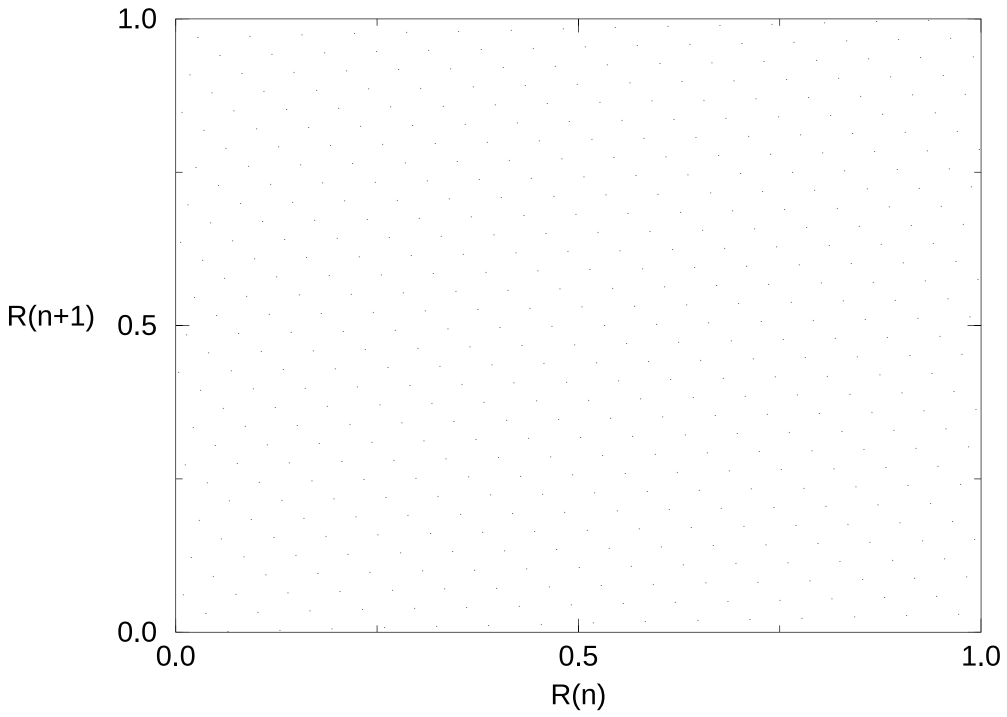

Figure 6: (c) Pair Plot of LCG(108,0,509,1)

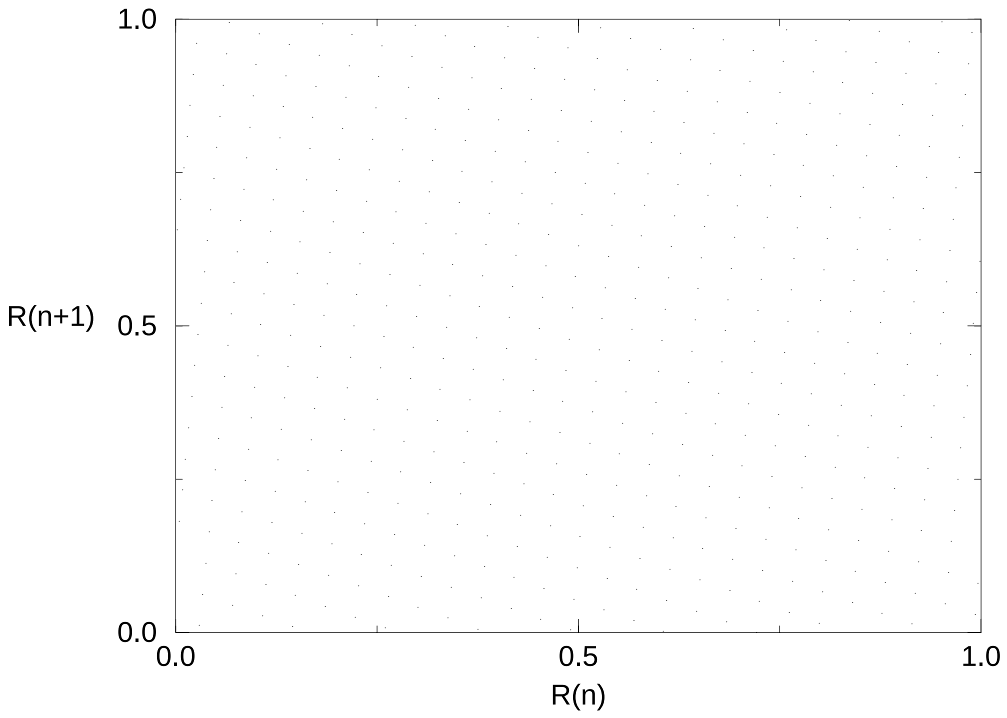

Figure 6: (d) Pair Plot of LCG(269,0,2048,1)

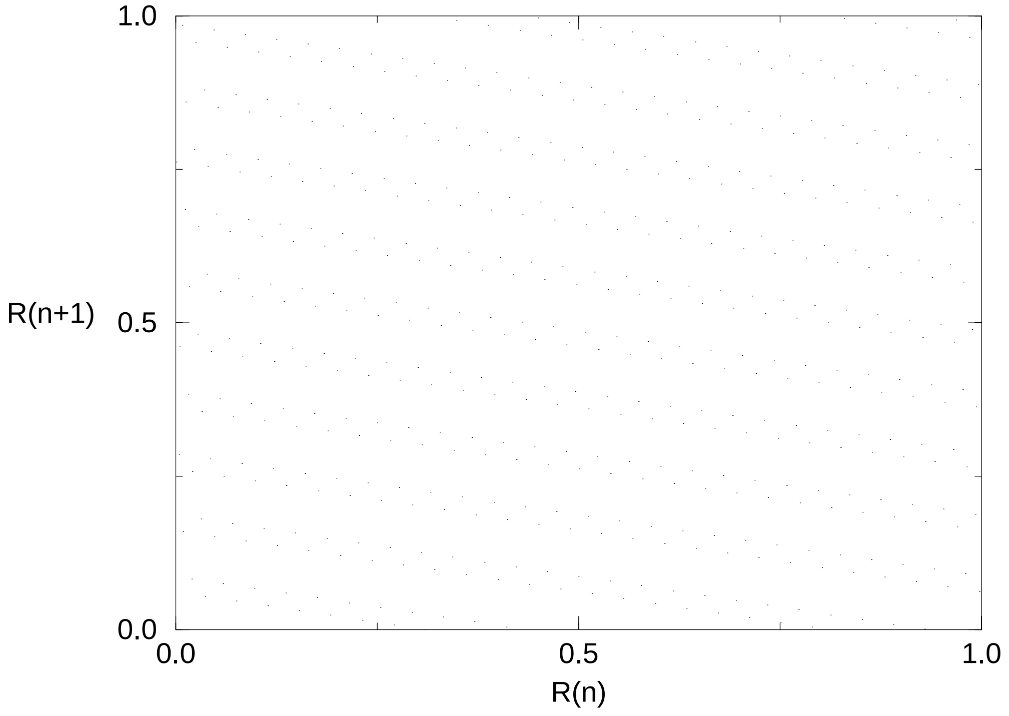

Figure 6: (e) Pair Plot of LCG(1203,0,2048,1)

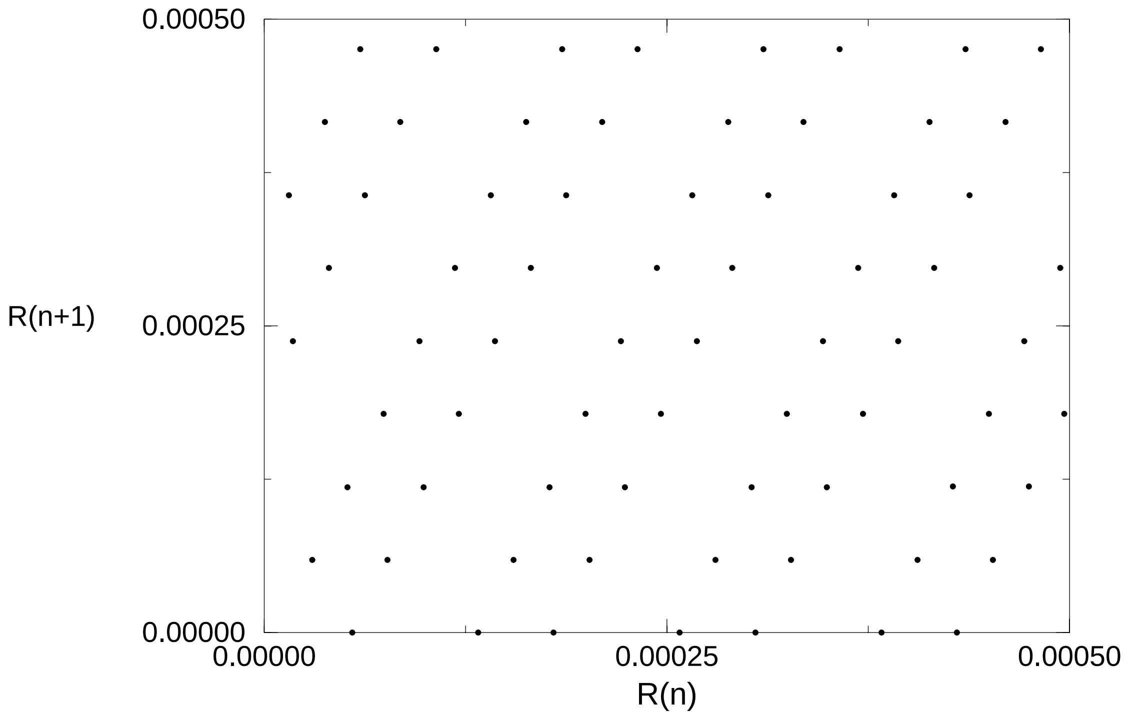

Figure 6: (f) Pair Plot of LCG(16807,0,2147483647,1)

The best situation is to have the maximum spacing of the rows, when viewed over all angles, as small as possible. When the maximum spacing is large, as it certainly is in Figure 6b, $LCG(128, 0, 509, 1)$, then clearly the unit square is not well covered by this set of points, and the results of our simulation may be adversely affected by this "striping." Figure 6c shows the effect of a good choice of a, $LCG(108, 0, 509, 1)$, for which the maximum spacing between rows is clearly smaller than it is in the first two cases, and where the points cover the unit square with nearly optimal uniformity, given that with this generator we can generate only a small number of points.

Figures 6a-c are examples that all use the prime modulus $m = 509$. When m is a power of two, the other major case to consider, the situation is similar, as depicted in Figures 6d and e. Recall that when m is a power of two and $c = 0$, the full period is $m/4$, but only if $a \equiv 3 \bmod 8$ or $a \equiv 5 \bmod 8$. Thus we choose $m = 2^{11} = 2,048$ so that with period 512, we plot almost the same number of points (508) as we see in Figures 6a-c. Figure 6d illustrates the result of using a good choice of a ($a = 269 \equiv 5 \bmod 8$). As in the prime modulus case, the plotted points form a uniform looking lattice. But when we plot the points in Figure 6e generated by taking $a = 1,203 \equiv 3 \bmod 8$, we see that the pattern looks like two lattice structures slightly offset from one another. This double lattice distinguishes the $a \equiv 3 \bmod 8$ case from the $a \equiv 5 \bmod 8$ case and is the reason that $a \equiv 5 \bmod 8$ is preferred, even though both sets of a values produce the same period. As an aside, the two lattice structures in the $a \equiv 3 \bmod 8$ case come about from plotting separately the pairs beginning with odd indices and those beginning with even indices.

The cases looked at so far are for moduli chosen to be small enough that we can plot the full-period results and still see a pattern with space between points. In practice, it would be foolish to use a generator with such a short period. If we look at an LCG with much longer period, one that we might use in a practical application, we still see the lattice-like full-period behavior. Figure 6f shows the full-period 2-D result for $LCG(16807, 0, 2^{31} - 1, 1)$, where we "zoom in" on only the portion of the unit square in the (0,0.0005) X (0,0.0005) corner, or 1/4,000,000 of the total area. Although the row spacing for this generator is obviously not optimal, this particular one is popular for its universal portability and its satisfactory performance in many applications.

We have shown only examples in which $c = 0$. Using a nonzero value for c when m is prime has the effect of shifting the entire lattice structure to a new location, so that it appears offset from that of the $c = 0$ case. The orientation and spacing of the lattice remain the same, except that one lattice point appears absent — this reflects the fact that when m is prime and $c = 0$, some nonzero value of X will succeed itself in the recursion formula of the generator. (Actually, there is a "hole" in the $c = 0$ lattice as well, but it lies on one of the axes, and so doesn't disrupt the apparent regularity.) When m is a power of two and c is nonzero, recall that the period of the generator is four times the period of the $c = 0$ case. Here again the resulting lattice appears to be aligned the same as the $c = 0$ lattice, with the $c = 0$ lattice offset from a subset of the $c \neq 0$ lattice. A further complication arises in the case where m is a power of two and $a \equiv 3 \bmod 8$: here a nonzero value for c will produce different offsets for each of the two distinct lattice structures, relative to the $c = 0$ case. The fact that the $c \neq 0$ case is slightly more complicated than the $c = 0$ case should not obscure the basic theme of this section: that the quality of an LGC used in generating n-tuples is a function of the multiplier, a.

The remarks given here with respect to 2-D behavior of LCGs apply as well to higher dimensions. For example, in three dimensions, the coordinate triples (R_1, R_2, R_3) formed as above will lie in distinct planes. For ease of illustration, we have considered only 2-D examples and only in a qualitative sense. The lattice spacing in any number of dimensions can be treated quantitatively using methods detailed in Knuth, Vol. 2. The interested reader is encouraged to pursue the topic further in this source. Remember that the effects described here are full-period properties of LCGs. If only a tiny fraction of the period is used, the most desirable way to use any pseudo-random generator, then these effects may not be noticed. If a user suspects that the lattice structure of n-tuple generation is affecting the outcome of a Monte Carlo simulation, then the best policy is to try another generator of different type and compare the results.

Exercise 7 - visualization of some poor LCGs

Modify the `ranlc.f` code to output (x, y) pairs of numbers as (R_1, R_2), (R_2, R_3), (R_3, R_4), etc. Plot these discrete points for the following cases:

(a) $a = 66$, $c = 0$, $m = 2^{31}$, 4,095 points (4,096 random numbers)
(b) $a = 5$, $c = 0$, $m = 2^{31} - 1$, 4,095 points (4.096 random numbers)

5 Vectorization and Access via Multiple Processors: LCGs

Many Monte Carlo applications have characteristics that make them easy to map onto computers having multiple processors. Some of these parallel implementations require little or no interprocessor communication (such applications are called "embarrassingly parallel") and are typically easy to code on a parallel machine. Others require frequent communication and synchronization among processors and in general are more difficult to write and debug. In developing any parallel Monte Carlo code, it is important to be able to reproduce runs exactly in order to trace program execution. Processors in MIMD machines are subject to asynchronous and unbalanced external effects and are thus, for all practical purposes, impossible to keep aligned in time in a predictable way. If the assumption is made that random number generation from a single generator will occur, across processors, in a certain predictable order, then that assumption will quite likely be wrong. A number of techniques have been developed that guarantee reproducibility in multiprocessor settings and with various types of Monte Carlo problems. We will consider only simple extensions to our previous discussion of LCGs, but acknowledge that there are many approaches to parallel random number generation in the literature. The first situation we address involves using LCGs in a fixed number of MIMD processes, where that number is known at the beginning of a run.

Suppose we know in advance that we will have N independent processes and that we will need N independent streams of random numbers. Then the best strategy for using an LCG is to split its period into nonoverlapping segments, each of which will be accessed by a different process. This amounts to finding N seeds which are known to be far apart on the cycle produced by the LCG. To find such seeds, first consider (for $c = 0$), the LCG rule successively applied:

$$X_{n+1} = aX_n \pmod m;$$

$$X_{n+2} = aX_{n+1} = a^2 X_n \pmod m;$$

$$X_{n+3} = aX_{n+2} = a^3 X_n \pmod m;$$

$$\ldots$$

$$X_{n+k} = aX_{n+k} = a^k X_n \pmod m;$$

Thus we can "leap ahead" k places of the period by multiplying the current seed value by $a^k \bmod m$. For our purposes, we would like N starting seeds, spaced at roughly $k = P/N$ steps apart. Since k is likely to be quite large, it is not practical to compute a^k one step at a time. Instead we compute an $(L+1)$-long array, $\bar{d}$, the power-of-two powers of a:

$$d_0 = a$$

$$d_1 = {d_0}^2$$

$$d_2 = {d_1}^2$$

$$\ldots$$

$$d_L = {d_{L-1}}^2$$

where d_L is the largest power-of-two power of a that is still smaller than k. I.e. L is the integer part of the log (base 2) of a. For example, assume that $k = 91 = 1011011_2$ (very small, but big enough to show how it works). Then

$$\bar{d} = (a, a^2, a^4, a^8, a^{16}, a^{32}, a^{64})$$

and since $91 = 64 + 16 + 8 + 2 + 1$, then

$$a^{91} = a^{64} \times a^{16} \times a^8 \times a^2 \times a^1 = d_6 \times d_4 \times d_3 \times d_1 \times d_0$$

Thus for any k, $a^k = \Pi d_i$ for all i for which bit i in the base-two representation of k is a one. Therefore we can leap ahead by k cycle steps with no more than $log_2 k$ multiplies. Once a^k is computed, the N seeds can be determined by the procedure:

$$\text{Choose} \quad seed_1$$

$$seed_2 = a^k seed_1 \pmod{m}$$
$$seed_3 = a^k seed_2 \pmod{m}$$
$$\cdots$$
$$seed_N = a^k seed_{N-1} \pmod{m}$$

With these seeds, each of the N processes will generate random numbers from nearly equally-spaced points on the cycle. As long as no process needs more than k random numbers, a condition easily met for some applications, then no overlap will occur. Everything just said for MIMD processes applies equally well to SIMD programs, where the number of random number streams needed is (usually) known at run time.

The development of the leap ahead technique just described assumed that $c = 0$ in the LCG rule. For $c \neq 0$, Leap ahead can still be accomplished in a similar way, if one constructs the $log_2 k$-long array of partial sums of the form: $s_j = \sum_{i=0}^{j} a^i$ where, as before, j is a power of two. The details are left as an exercise for the reader.

The second and more difficult case to consider is when we do not know at the beginning of program execution how many processes (generators) we will need. The splitting of processes in such programs are data driven and in most cases occur as the result of prior Monte Carlo trials taken many steps earlier. The problem is to spawn new LCG seeds in a way that is both reproducible and which yields independent new streams.

Here we only mention a generalized approach that works within limits. Further details can be found in [Frederickson, et al., 1984] Consider an LGC with the property that each X has two successors, a "left" successor, X_L, and a "right" successor, X_R. These are generated as follows:

$$L(X) = X_L = a_L X + c_L \pmod{m} \quad \text{and}$$
$$R(X) = X_R = a_R X + c_R \pmod{m}$$

Figure 7 shows the action of these operations with respect to a starting seed X_0. Taken separately, the X_L and X_R sequences are simple LCGs that traverse the set $\{0, 1, 2, \ldots, m-1\}$ in different order. Alternatively (and the method in which these generators are typically used), the X_L rule produces a pseudo-random leap-ahead for the X_R sequence, thus deterministically producing a seed for a new, spawned, subsequence of the "right" cycle. With such a mechanism that uses only local information from a process, reproducibility can be established. Frederickson gives a formula for the selection of the constants in the succession rules that satisfies a particular independence criterion, given some constraints. The interested reader is referred to Frederickson, et al., 1984 for further enlightenment.

Exercise 8 - Vectorization of Cray's LCG

Complete Exercise 6 above to determine the parameters a and m (probably $m = 2^{48}$) of Cray's `ranf()`. Develop a vectorized version of `ranf()` by creating a vector of successive multiples of the coefficient a. For a vector length of 64, for example, you will need a vector of $(a, a^2, a^3, \ldots, a^{64})$. Race your vectorized version of `ranf()` vs. Cray's `ranf()`, and see

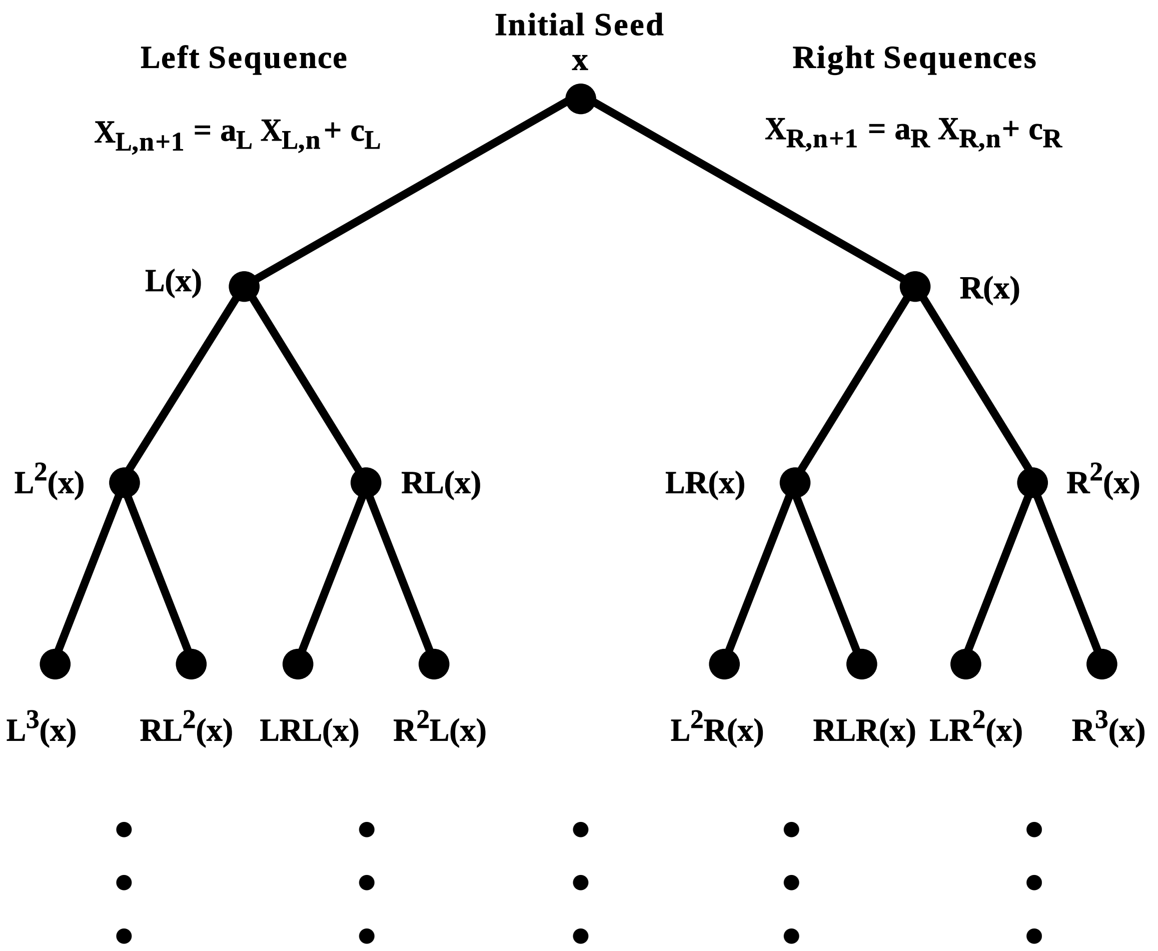

Figure 7: Tree Generated from ÒLeftÓ and ÒRightÓ Generator

how close you can come to their execution time. To do this, you will have to generate long vectors of random numbers, and use them elsewhere in the code. Use `second()` to obtain elapsed CPU time.

Exercise 9: Leap-ahead in an LCG

For the LCG, $X_{n+1} = aX_n + c$, where $c \neq 0$, determine $\hat{a}_k$ and $\hat{c}_k$ so that $X_{n+k} = \hat{a}_k X_n + \hat{c}_k$, resulting in a leap-ahead of k places on the original LCG cycle.

6 Lagged Fibonacci Generators

Lagged Fibonacci pseudo-random number generators have become increasingly popular in recent years. These generators are so named because of their similarity to the familiar

Fibonacci sequence:

$$1, 1, 2, 3, 5, 8, 13, \ldots \text{ defined by: } X_n = X_{n-1} + X_{n-2},$$

where the first two values, X_0 and X_1, must be supplied. This formula is generalized to give a family of pseudo-random number generators of the form:

$$X_n = X_{n-\ell} + X_{n-k} \pmod{m}; \quad \text{where } \ell > k > 0$$

and where, instead of two initial values, ℓ initial values, $X_0, \ldots, X_{\ell-1}$, are needed in order to compute the next sequence element. In this expression the "lags" are k and ℓ, so that the current value of X is determined by the value of X k places ago and ℓ places ago. In addition, for most applications of interest m is a power of two. That is, $m = 2^M$. (This type of pseudo-random number generator, along with several others, has been extensively tested for randomness properties by Marsaglia [Marsaglia, 1985] and has been given high marks. The only deficiency found was related to what he calls the Birthday Spacings test, for low values of ℓ and k. The interested reader is referred to Marsaglia, 1985.)

With proper choice of k, ℓ, and the first ℓ values of X, the period, P, of this generator is equal to $(2^\ell - 1) \times 2^{(M-1)}$. Proper choice of ℓ and k here means that the trinomial $x^\ell + x^k + 1$ is primitive over the integers mod 2. The only condition on the first ℓ values is that at least one of them must be odd. Using the notation $LFG(\ell, k, M)$ to indicate the lags and the power of two modulus, examples of two commonly used versions of these generators are:

$$1)\ LFG(17,\ 5, 31)\text{: } \ell = 17,\ k = 5,\ M = 31 \quad \Rightarrow P \approx 2^{47}$$

$$2)\ LFG(55, 24, 31)\text{: } \ell = 55,\ k = 24,\ M = 31 \quad \Rightarrow P \approx 2^{85}$$

Obviously, the value of the modulus, m, does not by itself limit the period of the generator, as it does in the case of an LCG. Note also that lagged Fibonacci pseudo-random number generation is computationally simple: an integer add, a logical AND (to accomplish the mod 2^M operation), and the decrementing of two array pointers are the only operations required to produce a new random number X. Furthermore, with a large enough value of k, limited vectorization can be achieved. The major drawback in the case of this type of generator is the fact that ℓ words of memory must be kept current. An LCG requires only one: the last value of X generated.

We now look at some of the theory of these generators, with a view toward ensuring their proper use. Conceptually, a Fibonacci generator acts the same as a linear shift register, and if we set $M = 1$ so that $m = 2^1$, then we have a binary linear shift register. Figure 8 is a diagram of a particular binary linear shift register, where $\ell = 10$, $k = 7$, and every X_j is either a 1 or a 0. (We will use this choice of ℓ and k in several examples in order to illustrate in a non-trivial way some of the properties of this type of generator.) The arrows are there to depict the motion of the shift register as it makes the transition from one state to the next. This is called advancing the register. Two such advances are shown in Figure 9.

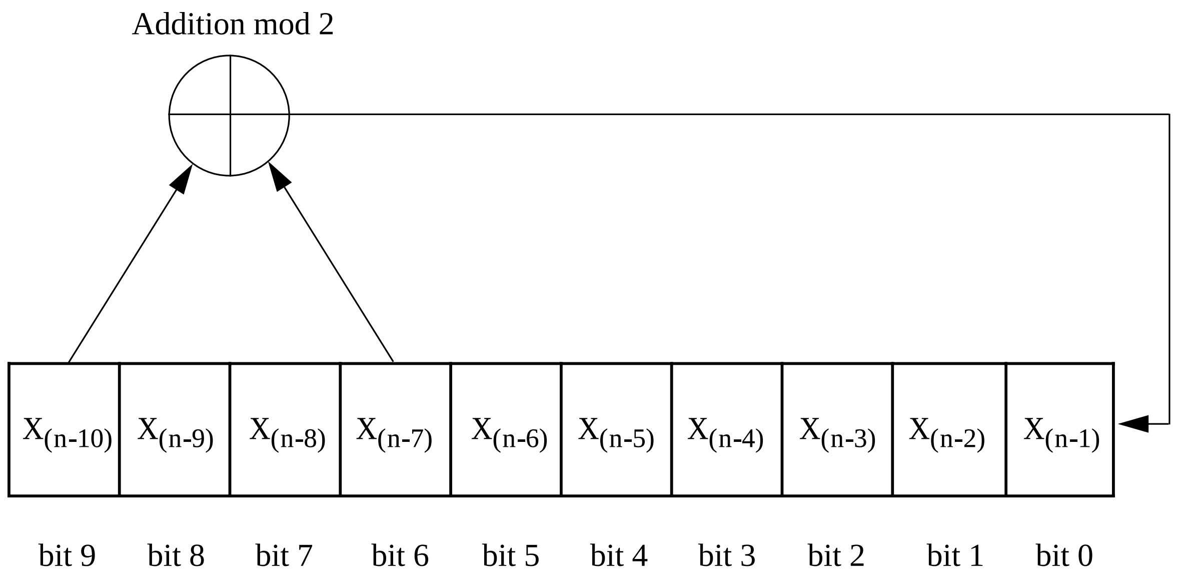

Figure 8: State Transition Diagram for Binary Shift Register

If we set m equal to a higher power of two, say $m = 2^4$, i.e., $M = 4$, then the single-bit values of the X's in Figure 8 will instead be represented by M bits each. Figure 10 is a diagram of the mod-16 linear shift register associated with the equation

$$X_n = X_{n-10} + X_{n-7} \pmod{2^4}$$

In looking at this more general Fibonacci generator, $LFG(10, 7, 4)$, two things are worth noting:

1. If we look for a moment at only the least significant bits of the elements in this register, that is, only those bits in the bottom row, then we see that the behavior of this row is unaffected by bits in the higher rows. The contents of the bottom row will therefore be indistinguishable from those of the binary shift register in Figure 8.

   This is no surprise — when we add two numbers, the answer in the one's place does not depend in any way on the digits in the ten's place (here, the two's place).

2. If the bottom row of bits is all zeros, then no amount of register motion can produce any ones there. This is to say that if the first ℓ values of X are all chosen to be even, then there can be no subsequent X values that are odd when the register is advanced. In such a case the pseudo-random number generator will not be of full period. The period can be no larger — and may be smaller — than $P/2$, since all odd numbers will be excluded.

Since the state transformation of the shift register contents is a linear operation, a matrix equation describing it can be given. Continuing with the example of the 10-long generator,

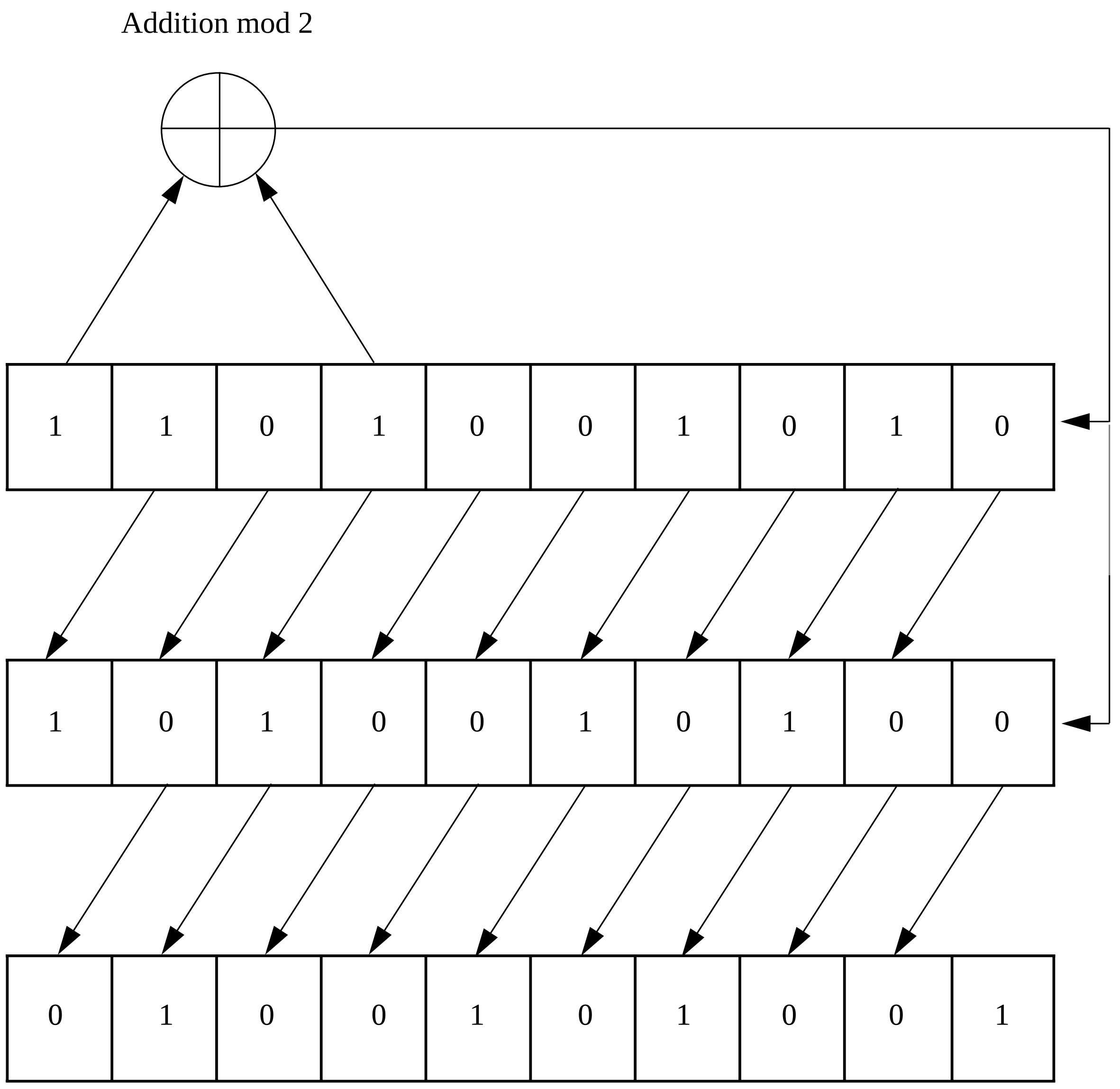

Figure 9: XXX

if we define $\mathbf{x}$ and $\mathbf{A}$ by:

$$\mathbf{x} = \begin{pmatrix} X_{n-1} \\ X_{n-2} \\ X_{n-3} \\ X_{n-4} \\ X_{n-5} \\ X_{n-6} \\ X_{n-7} \\ X_{n-8} \\ X_{n-9} \\ X_{n-10} \end{pmatrix} \quad \text{and} \quad \mathbf{A} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix},$$

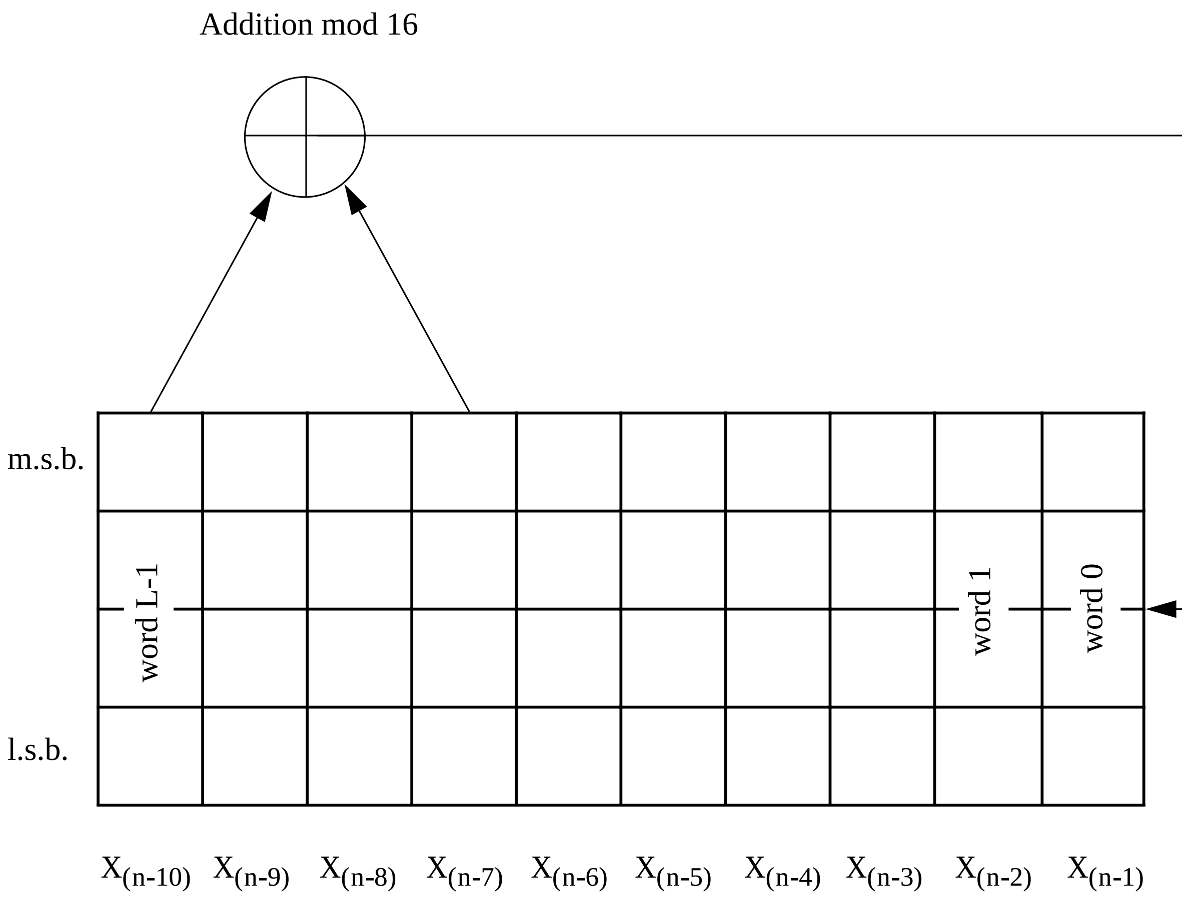
Addition mod 16
m.s.b.
l.s.b.
word L-1
word 1
word 0
X(n-10) X(n-9) X(n-8) X(n-7) X(n-6) X(n-5) X(n-4) X(n-3) X(n-2) X(n-1)

Figure 10: Register Motion for LFG(10,7,4)

then the action of the shift register can be readily described by the equation

$$\mathbf{x}_n = \mathbf{A}\mathbf{x}_{n-1} \pmod{2^M},$$

where $\mathbf{x}_n$ is the entire vector after n time steps. If the vector $\mathbf{x}_0$ has been given some initial set of values, then we have,

$$\mathbf{x}_1 = \mathbf{A}\mathbf{x}_0 \ ;$$

$$\mathbf{x}_2 = \mathbf{A}\mathbf{x}_1 = \mathbf{A}^2\mathbf{x}_0 \ ;$$

$$\mathbf{x}_3 = \mathbf{A}\mathbf{x}_2 = \mathbf{A}^2\mathbf{x}_1 = \mathbf{A}^3\mathbf{x}_0 \ ;$$

$$\text{etc.}$$

and in general,

$$\mathbf{x}_n = \mathbf{A}^n\mathbf{x}_0 \ .$$

As an aside, note that $\mathbf{x}_P = \mathbf{A}^P\mathbf{x}_0 = \mathbf{x}_P$, or $\mathbf{A}^P = I$, the identity matrix.

The above equation gives a way to leap the generator ahead, similar in fashion to the leap ahead concept discussed for LCGs. But such an operation with a Fibonacci generator requires a matrix-vector multiply involving, at best, the precomputation of possibly many multiples of $\mathbf{A}$. This precomputation may be expensive, even for small values of ℓ (recall that $\mathbf{A}$ is $\ell \times \ell$), and may be prohibitive for large values of ℓ. Leaping a Fibonacci generator ahead is therefore not recommended. This does not mean that such a generator cannot be "split," however. We now look at an efficient method for splitting a Fibonacci generator.

Before explaining the splitting technique, notice that the period, P, of a properly constructed Fibonacci generator is $(2^\ell - 1)2^{M-1}$. Consider a given initial set of values of an M-bit, ℓ-long Fibonacci register. This state is a particular $(M \times \ell)$ bit pattern in the rectangular register. If the register is advanced P times, the initial pattern will be replaced by $P - 1$ different patterns before the initial one reappears. But the number of *possible* bit patterns in the register is $2^{M\times\ell}$, a number far larger than P. This tells us that there are many P-long cycles that are independent of one other and can be generated from the same $LFG(\ell, k, M)$ structure. The number of such full-period cycles is $2^{(\ell-1)(M-1)}$ [Mascagni, et al., 1994]. For example, with the $LFG(10, 7, 4)$ generator of our previous example, there are 2^{27} separate full-period cycles and a much smaller number of less than full-period cycles $(2^{18} + 2^9 + 1$, to be precise).

The question now becomes, how do we initialize separate cycles? This problem is addressed by Mascagni, et al. [Mascagni, et al., 1994], where they describe a canonical form for initializing Fibonacci generators. This canonical form is determined by ℓ and k, but is independent of M. To understand the use of this canonical form, consider a second view of the $LFG(10, 7, 4)$ register shown in Figure 11. The L-shaped region along the left column and the bottom row is fixed with all zeros, except for a one in the least significant bit of word 7 (the word associated with X_{n-8}). The remaining bits, those in the (9×3)-bit rectangular region filled with z's, are free to be chosen in any combination. Each combination of bits in the (9×3) area will generate a distinct cycle of pseudo-random numbers. In other words, every possible bit pattern that can be put in the canonical form will occur in one and only one full period cycle.

In general, the canonical form for initializing Fibonacci generators requires that word $(\ell - 1)$ be set to all zero bits and that the least significant bits of all words in the register to be set to zero, with the exception of one or two characteristic bits that depend on ℓ and k. As shown in Figure 11, the canonical form for the $LFG(10, 7, 4)$ generator requires the least significant bit of word 7 to be set to one, with all other least significant bits in the register set to zero. Table 4 lists, for several choices of $LFG(\ell, k, M)$, the characteristic word (or words) for which the least significant bit should be set to one in order to be in canonical form.

Two caveats with respect to the use of this canonical form should be mentioned at this point. Suppose a user were to construct and initialize a number of generators using an initialization scheme that simply numbered the initial states as one would number the integers in a normal binary representation. This method certainly would not violate the conditions under which the canonical form produces distinct cycles, and indeed the cycles of random numbers generated from each initial state would be different. But the first few random numbers from these cycles may be less than random-looking when compared both with corresponding

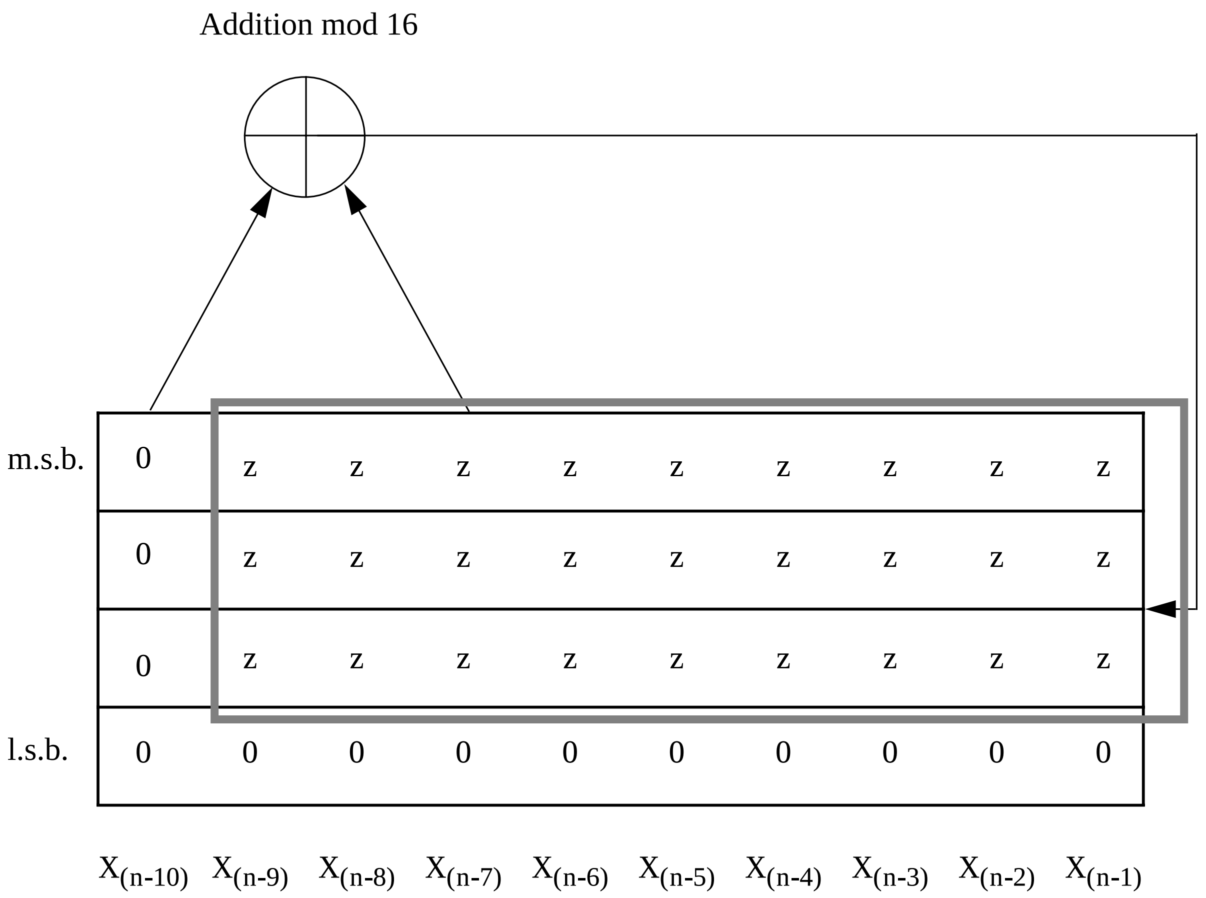
Addition mod 16
m.s.b.
l.s.b.
0 z z z z z z z z z
0 z z z z z z z z z
0 z z z z z z z z z
0 0 0 0 0 0 0 0 0 0
$X_{(n-10)}$ $X_{(n-9)}$ $X_{(n-8)}$ $X_{(n-7)}$ $X_{(n-6)}$ $X_{(n-5)}$ $X_{(n-4)}$ $X_{(n-3)}$ $X_{(n-2)}$ $X_{(n-1)}$

Figure 11: Canonical Form for LFG(10,7,4)

output from other generators and with successive values from the same generator. As an example, consider the $LFG(10, 7, 4)$ generator, where we canonically initialize one register with all zeros in the z-ed region of Figure 11 (call this generator “A”) and a second generator with a single one in the lower right corner of the z-ed region (call this generator “B”). Table 5 lists the first 100 pseudo-random numbers produced by both of these generators, both as integers in the range [0,15] and as floating point numbers in the range [0.0,1.0). Note that the A sequence consists of low numbers (less than 0.5) for 43 iterations and is actually either zero or one for the first 19 iterations. B is somewhat better, but still not very satisfying. In addition, A and B appear, at least until about 50 iterations, to be correlated. This appearance of non-random behavior on the part of these two sample sequences does not mean that Fibonacci sequences or this method of initializing them does not produce high quality random numbers. “Flat spots” are to be expected in any good pseudo-random number sequence of sufficient length, and indeed such a sequence without them would be suspect with regard to its randomness. The problem with initializing A and B in the example is that the flat spots were “lined up” with each other and were placed at the very beginning. Neither of these results would be a problem if our application required the use of thousands of numbers

Table 4: Canonical Form Specifications for Selected Fibonacci Generators

ℓ	k	word no.(s) for lsb = 1
3	2	0
5	3	1,2
10	7	7
17	5	10
35	2	0
55	24	11
71	65	1
93	91	1,2
127	97	21
158	128	63

from both the A and the B sequences and if the numbers generated at the beginning were no more important than those generated later. After a short time, the two sequences would look completely uncorrelated from each other and would appear internally random as well. However, if the number of required pseudo-random numbers is small, the outcome of the numerical experiment might be unexpectedly skewed due to the initial conditions.

The problem just demonstrated is not difficult to fix. We simply need a different way of "numbering" the patterns in the area of the free bits in the canonical form, one that randomizes these patterns to some extent. An LCG can be used to initialize the free bits, if one is careful to use it in such a way that a unique bit pattern results for each distinct cycle number number. Pryor et al. [Pryor et al., 1994] describe the use of the LCG of Park and Miller [Park and Miller, 1988] as a good method for initializing their family of Fibonacci generators. In that family, M is equal to 32, so that the free bit rectangle is 31 bits high; whereas the Park and Miller generator uses a modulus of the prime number $2^{31} - 1$. Thus the successive values in the 31 high bits of the n^{th} initial state for word 0 through word $(\ell - 2)$ are simply the $(\ell - 1)$ LCG successors of n. This gives a simple way to initialize $2^{31} - 1$ distinct cycles that start out with a satisfactory "look and feel." As an aside, this particular LCG passes the 'bitwise" randomness tests described by Altman [Altman, 1988], for use in initializing Fibonacci generators.

The second caveat related to the use of this canonical form is also highlighted by the previous example. The astute reader may have noticed that with this method of initializing the Fibonacci register, for every advance, the least significant bit is the same for all generators. Unfortunately, this is the tradeoff vs. efficiency that was made in order to guarantee uniqueness of the cycles: the least significant bit is simply non-random relative to the other

Table 5: The First 100 Random Numbers from LFGs “A” and “B”

	X_n		R_n			X_n		R_n	
n	Seq. “A”	Seq. “B”	Seq. “A”	Seq. “B”	n	Seq. “A”	Seq. “B”	Seq. “A”	Seq. “B”
1	0	0	0.000	0.000	51	15	7	0.938	0.438
2	0	0	0.000	0.000	52	1	15	0.062	0.938
3	1	1	0.062	0.062	53	1	1	0.062	0.062
4	0	0	0.000	0.000	54	4	2	0.250	0.125
5	0	0	0.000	0.000	55	7	1	0.438	0.062
6	0	0	0.000	0.000	56	0	2	0.000	0.125
7	0	2	0.000	0.125	57	15	11	0.938	0.688
8	0	0	0.000	0.000	58	5	11	0.312	0.688
9	0	0	0.000	0.000	59	1	1	0.062	0.062
10	1	3	0.062	0.188	60	6	8	0.375	0.500
11	0	0	0.000	0.000	61	3	9	0.188	0.562
12	0	0	0.000	0.000	62	8	0	0.500	0.000
13	1	1	0.062	0.062	63	1	3	0.062	0.188
14	0	2	0.000	0.125	64	3	13	0.188	0.812
15	0	0	0.000	0.000	65	12	12	0.750	0.750
16	0	0	0.000	0.000	66	1	3	0.062	0.188
17	1	5	0.062	0.312	67	5	3	0.312	0.188
18	0	0	0.000	0.000	68	8	4	0.500	0.250
19	0	0	0.000	0.000	69	9	1	0.562	0.062
20	2	4	0.125	0.250	70	7	11	0.438	0.688
21	0	2	0.000	0.125	71	6	6	0.375	0.375
22	0	0	0.000	0.000	72	4	12	0.250	0.750
23	1	1	0.062	0.062	73	2	6	0.125	0.375
24	1	7	0.062	0.438	74	8	0	0.500	0.000
25	0	0	0.000	0.000	75	4	0	0.250	0.000
26	0	0	0.000	0.000	76	10	4	0.625	0.250
27	3	9	0.188	0.562	77	12	14	0.750	0.875
28	0	2	0.000	0.125	78	14	10	0.875	0.625
29	0	0	0.000	0.000	79	13	13	0.812	0.812
30	3	5	0.188	0.312	80	9	1	0.562	0.062
31	1	9	0.062	0.562	81	14	6	0.875	0.375
32	0	0	0.000	0.000	82	8	12	0.500	0.750
33	1	1	0.062	0.062	83	12	10	0.750	0.625
34	4	0	0.250	0.000	84	4	14	0.250	0.875
35	0	2	0.000	0.125	85	2	10	0.125	0.625
36	0	0	0.000	0.000	86	7	1	0.438	0.062

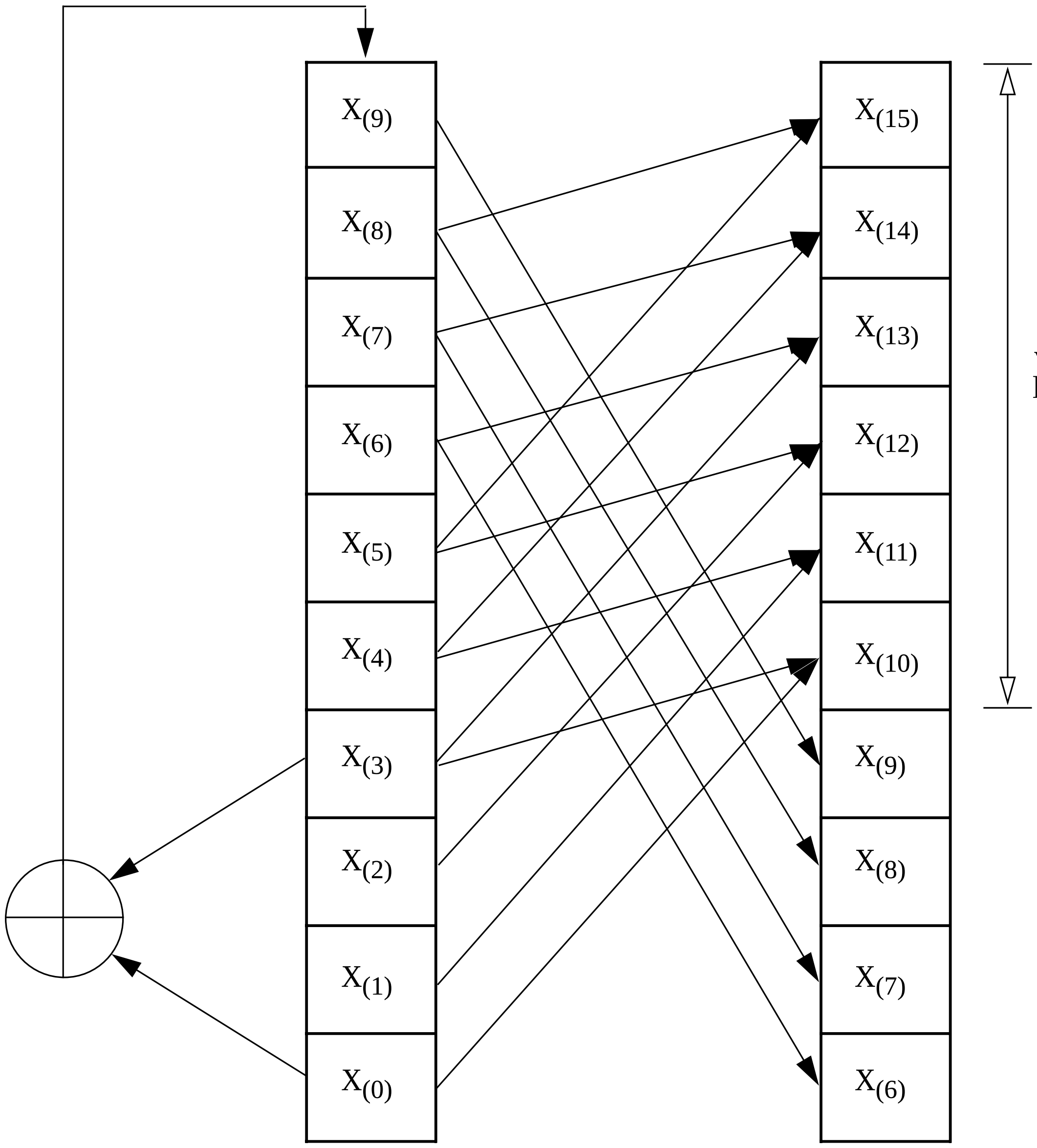

Figure 12: Vectorized Stepping of LFG(10,7,M)

cycles. For all other bit fields, including the next-to-least significant, there is no such defect. Several possibilities arise for remedying this situation, including the use of a separate, independent generator for only the least significant bit. However, in the interest of simplicity and speed, Pryor et al. have chosen to shift off the least significant bit of the generated random number, so that the numbers returned by the generator are in the range $[0, 2^{31} - 1]$ rather than $[0, 2^{32} - 1]$. The register is initialized and maintained as a 32-bit generator, so that no loss of period length or uniqueness of cycle is incurred. And for 32-bit machines, the returned results still include all positive integers.

As mentioned in the beginning of this section, there is at least limited potential for vectorization of a single lagged Fibonacci generator. Figure 12 is an illustration of vectorization applied to our $LFG(10, 7, M)$ generator. As the figure shows, the vector algorithm advances the register ahead by $(k - 1)$ steps, so that the vector length of most of the operations is $(k - 1)$. Note that there is a vector copy operation of length $(\ell - (k - 1))$. Care should be taken that no item of data is destroyed before it is needed. The easiest way to prevent unintentionally writing over needed data is to keep two copies of the Fibonacci register and, for each "vector" advance, use the old copy to construct the new one. None of the data in the old copy will be destroyed until the next vector advance, when it becomes the new copy. If vectorization of the Fibonacci generator is important — and it could be, if random number generation consumes a large fraction of the execution time — then clearly a long vector length is better than a short one. Processing with a vector length of 6, as our example has, would not yield much improvement over the scalar method. For vectorization to provide meaningful improvement over scalar processing, the vector operations should be long enough to make good use of the machine hardware. For example, on Cray machines where the vector registers are 64 words long (128 on the new models), this usually means vector lengths of tens of elements. For these machines, the generators $LFG(71, 65, M)$ and $LFG(159, 128, M)$ would be good choices, with respective vector lengths of 64 and 127.

In Figure 13 we list a sample Fortran code for initializing and generating random numbers from the generator $LFG(17, 5, 32)$. Note that the register is maintained as a set of 32-bit numbers, but that the number returned to the user has only 31 bits. The initialization of the register is accomplished using the Park and Miller LGC described in [Park and Miller, 1988]. The seed, "`iseed0`," supplied by the user may be any integer greater than or equal to zero and less than or equal to $2^{31} - 2 = 2{,}147{,}483{,}646$. The register is initialized in canonical form, so each value of `iseed0` results in a distinct cycle of random numbers. Since the function `irnd175()` was written to work on 64-bit machines, as well as 32-bit machines, the mask operations were included to add clarity to the code. In many situations, the 32-bit mask operation could be eliminated, since the hardware would simply ignore any overflow. The 31-bit mask could also be eliminated on any systems that zero-fill on right shift operations. If the system performs a "sign extension" type of fill, then the 31-bit mask would be required.

Exercise 10 - to explore LFGs visually

Use the code `ranlf.f` to generate random numbers. Modify the code to print out pairs of random numbers on [0,1) and display them as points as R_{n+1} vs. R_n for 4,095 points (4,096 random numbers).

(a) Using $\ell = 17$ (n varies from 0 to 16), and $k = 12$, select a subtractive generator. Use an initial seed of 37 to generate the initial state. Compare the display to those of Figures 6a-f in Section 4 which were generated using LCG generators.

(b) Select different values for k, but keep ℓ fixed at 17. Use the parameters as in (a) above. Generate plots as above in (a). What do you observe about the uniformity of the random numbers and their correlation?

```
      call init175(iseed0)
         . . .
      krand = irnd()
      krand = irnd()
      krand = irnd()
         . . .
c
c
      subroutine init175(iseed)
      implicit integer (a-z)
      parameter(a = 16807, r = 2836, q = 127773, m = z'7fffffff')
      common/c175/ ifibreg(17),lptr,kptr
c
      do  i=1,16
         hi                   = iseed/q
         lo                   = mod(iseed,q)
         iseed                = a*lo - r*hi
         if(iseed.lt.0) iseed = iseed + m
         ifibreg(i)           = ishft(iseed,1)
      enddo
      ifibreg(11) = ifibreg(11) + 1
      lptr        = 17
      kptr        = 5
      return
      end
c
c
      integer function irnd175()
      parameter(mask32 = z'ffffffff',  mask31 = z'7fffffff')
      common/c175/ ifibreg(17),lptr,kptr
c
      itemp              = ifibreg(lptr) + ifibreg(kptr)
      ifibreg(lptr)      = iand(itemp,mask32)
      lptr               = lptr - 1
      kptr               = kptr - 1
      if(lptr.le.0) lptr = 17
      if(kptr.le.0) kptr = 17
      irnd175            = iand(ishft(itemp,-1),mask31)
      return
      end
```

Figure 13 FORTRAN implementation of $LFG(17, 5, 32)$

Exercise 11 - the fundamental concepts of LFGs

Consider the Fibonacci generator $LFG(3,2,2)$.

(a) Draw a shift register diagram, similar to Figure 10, to represent , the action of this generator.

(b) How long is the period, P, of this generator?

(c) How many distinct cycles does this generator have?

(d) Using the information for this generator in Table 4, initialize, in canonical form, every cycle for this generator, and generate each full cycle.

(e) Suppose the generator $LFG(3,2,2)$ were initialized with the values, $X_1 = 1$, $X_2 = 3$, and $X_3 = 1$. If the generator were advanced for a full cycle, which of the "canonical" cycles generated in part (d) would be the same as this new one? How many advances of the generator did you need to determine this?

(f) Would parts (a) through (e) be fun to do if the generator changed only slightly, from $LFG(3,2,2)$ to $LFG(3,2,3)$? What is the value of P for $LFG(3,2,3)$? How many distinct cycles does it have?

Exercise 12 - various LFGs as they influence the number and length of cycles

Using a computer and your favorite programming language, go through steps (a) through (e) as in Exercise 11, above, for the generator $LFG(5,3,2)$. What about part (f), i.e., changing $LFG(5,3,2)$ to $LFG(5,3,3)$? What would that change do to the number of cycles and to P?

7 Summary and Recommendations

We have illustrated through example applications the implementation of various random number generators. We have concerned ourselves much more with issues of portability and quality than with efficiency, as our observations have been that generating random numbers is not very consumptive of CPU resources, except in a narrow range of circumstances. Our observations have led us to conclude that, now, due to more powerful computers, more sophisticated physical models are employed. This results in codes which do more work per random trial than in the past.

Although this is not a formal treatment, we have been able to infer that there exist good random number generators for 32 bit machines. The portable generator recommended by Park and Miller [Park and Miller, 1988] appears to be particularly suitable for "ordinary" problems. At very large scale, this random number generator's period of $2^{31} - 2$ (over 2 billion) may be too short.

Using lagged Fibonacci generators shows great promise, and is now undergoing thorough testing — particularly in terms of correlations of sequences. It appears particularly suitable for generating many streams of independent random numbers in parallel. The interested reader should “stay tuned” for further developments. Moreover, we believe these generators are now sufficiently mature that we recommend them for inclusion in existing applications. The quality of the random sequences are at least as good as those generated using LCGs, with the only drawback being the requirement for greater storage. In cases requiring *many* parallel generators, this can be a factor.

Finally, we wax philosophical. We urge the supercomputist to approach the generation of random numbers with circumspection, particularly when solving very large-scale problems. All random number generators should be tested thoroughly for their quality before being used upon other than academic problems. Further, at least two levels of quality in random number generators should be implemented to assess whether the answers are independent of the random numbers. This is a matter requiring judgment and is aided by experience.

However, the situation is not as bleak as it may seem from the above discussion. For very large scale problems lacking a regular structure (e.g., complex geometries and/or material properties), it is very unlikely that correlations between random numbers generated on separate processors will be significant due to the fact that the physical problem exhibits a great many possibilities for random events. Further, if the parameters of the generator are chosen correctly, there are a great many possibilities for random events when using random real numbers. However, the truly cautious researcher should not rely on the confluence of these factors to discourage assessing the results in the context of quality of the random events derived from the random number generator.

Further, it is wise to mention that the above observations are specific to using random real numbers, and not random integers. In particular, there may exist a resonance between the physical problem and random integers such as the successive odd/even pairs obtained when using a modulus of a power of two in a linear, congruential generator. As the complexity of the problem decreases, the potential for problems with the random number generator increases.

We note that FORTRAN 90 will obviate many of the impediments to a portable implementation of random number generators, as: (1) it allows arbitrary precision in numbers, which can be made large enough to eliminate overflow and problems with a sign bit, and (2) it has embedded in the language functions to return the date and time, which now are system calls and vary from system to system. Moreover, a random number generator intrinsic function exists as part of the language, and it may be quite good (although it will have to be tested thoroughly by the Monte Carlo community before gaining universal acceptance).

8 References

N. S. Altman. “Bitwise Behavior of Random Number Generators,” *SIAM J. Sci. Stat. Comput., 9(5)*, September, pps. 941-949, 1988. G. M. Amdahl. “Validity of the single processor approach to achieving large-scale computing capabilities,” *Proceedings of the American Federation of Information Processing Societies, 30*, Washington, DC, pps. 483-485, 1967.

S. L. Anderson. “Random Number Generators on Vector Supercomputers and Other

Advanced Architectures," *SIAM Review, 32 (2)*, pps. 221-251, 1990.

E. F. Beckenback (ed.) *Modern Mathematics for the Engineer*, McGraw Hill, New York, NY, 1956.

K. Binder. *Applications of the Monte Carlo Method in Statistical Physics*, Springer-Verlag, Berlin, 1984.

J. Briesmeister, ed. "MCNP: a general Monte Carlo code for neutron and photon transport," LA-7396-M, Rev 2, Los Alamos National Laboratory report, 1986.

T. B. Brown. *Vectorized Monte Carlo*, Ph.D. Dissertation, Department of Nuclear Engineering, University of Michigan, 1981.

C. E. Burghart and P. N. Stevens. "A general method of importance sampling the angle of scattering in Monte Carlo calculations," *Nuclear Science and Engineering 46(1)*, 12- 21, 1971.

P. Burns, M. Christon, R. Schweitzer, H. Wasserman, M. Simmons, O. Lubeck and D. Pryor. "Vectorization of Monte Carlo particle transport - an architectural study using the LANL benchmark GAMTEB," *Proceedings, Supercomputing '89*, Reno, NV, 10 - 20, Nov. 13, 1988.

P. J. Burns and D. V. Pryor. "Vector and parallel considerations for the Rayleigh problem in molecular gas dynamics," *Proceedings, 7th International Conference on Finite Element Methods in Flow Problems*, Huntsville, AL, April 3-7, 1989.

P. J. Burns and D. V. Pryor. "Vector and parallel Monte Carlo radiative heat transfer," *Numerical Heat Transfer, Part B: Fundamentals 16(101)*, 1989.

R. D. Chandler, J. N. Panaia, R. B. Stevens and G. E. Zinmeister. "The solution of steady-state convection problems by the fixed random walk method," *Journal of Heat Transfer 90(3)*, 361-363, 1968.

Cray Research, Inc. UNICOS Libraries, Macros and Opdefs Reference Manual, Publication SR-2013, Cray Research, Inc., Mendota Heights, MN, 1987.

D. Crockett, J. D. Maltby and P. J. Burns, "MONT3V user's manual," Internal Publication, Department of Mechanical Engineering, Colorado State University, 1989.

J. H. Curtis. "Sampling Methods Applied to Differential and Difference Equations," *Proceedings IBM Seminar on Scientific Computation*, Nov. 1949, IBM Corp., New York, 87-109, 1950.

L. Dagum. "Implementation of a hypersonic rarefied flow particle simulation on the connection machine," *Proceedings, Supercomputing '89*, ACM Press, Baltimore, MD, 42-50, Nov. 13-17, 1989.

R. Eckhard. "Stan Ulam, John von Neumann and the Monte Carlo method," *Los Alamos Science, 15*, 1987.

L. W. Ehrlich. "Monte Carlo Solutions of Boundary Value problems Involving the Difference Analogue of $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}$," *Journal of the Association of Computing Machinery, 6*,204-218, 1959.

A. F. Emery and W. W. Carson. "A modification to the Monte Carlo method - the exodus method," *Journal of Heat Transfer 90(3)*, 328-332, 1968.

P. Frederickson, R. Hiromoto, T. L. Jordan, B. Smith, T. Warnock, "Pseudo-random trees in Monte Carlo," *Parallel Computing, 1(3)*, December, 175-180, 1984.

S. W. Golomb, *Shift Register Sequences*, (Revised Ed.), Aegean Park Press, 1982.

A. Haji-Sheikh, "Monte Carlo Methods," Ch. 16 in *Handbook of Numerical Heat Transfer*, 673-722, 1988.

A. Haji-Sheikh and E. M. Sparrow. "The Floating Random Walk and its Application to Monte Carlo Solutions of Heat Equations," *Journal of Heat Transfer 89(2)*, 121-131, 1967.

J. M. Hammersly and D. C. Handscomb. *Monte Carlo Methods*, Methuen, London, 1964.

D. B. Heifetz. "Vectorizing and macrotasking Monte Carlo neutral particle algorithms," Princeton Plasma Physics Laboratory Report PPPL-2427, April 1987.

J. R. Howell. Application of Monte Carlo to Heat Transfer Problems, *Advances in Heat Transfer, 5*, 1-54, 1968.

C. C. Hurd, ed., *Proceedings of the Seminar on Scientific Computation*, International Business Machines Corporation, New York, NY, Nov. 1949.

H. Kahn and A. W. Marshall. "Methods of reducing sample size in Monte Carlo computations," *Journal of Operations Research Society of America, 1*, 263-278, 1953.

M. H. Kalos. "Monte Carlo methods and the computers of the future," Ultracomputer Note #83, April 1985.

M. A. Kalos and P. A. Whitlock, *Monte Carlo Methods, Volume I: Basics*, Wiley Interscience, New York, 1986.

J. P. C. Kleinjnen. *Statistical Techniques in Simulation, Part 1*, Marcel Dekker, New York, NY, 1974.

D. E. Knuth. Seminumerical Algorithms, 2nd Ed., Vol. 2 of *The Art of Computer Programming*, Addison-Wesley, Reading PA, 1981.

J. M. Lanore. "Weighting and biasing of a Monte Carlo calculation for very deep penetration of radiation," *Nuclear Science Engineering 45(1)*, 66-72, 1971.

M. E. Larsen and J. R. Howell. "Least-squares smoothing of direct-exchange areas in zonal analysis," *Journal of Heat Transfer, 108*, 239-242, 1986.

J. D. Maltby. *Three-dimensional Simulation of Radiative Heat Transfer by the Monte Carlo Method*, Master's Thesis, Department of Mechanical Engineering, Colorado State University, 1987.

J. D. Maltby. *Analysis of Electron Heat Transfer via Monte Carlo Simulation*, Ph.D. Dissertation, Department of Mechanical Engineering, Colorado State University, 1987.

J. D. Maltby and P. J. Burns, "MONT2D and MONT3D user's manual," Internal Publication, Department of Mechanical Engineering, Colorado State University, 1988.

J. D. Maltby and P. J. Burns, "MONT3E user's manual," Internal Publication, Department of Mechanical Engineering, Colorado State University, 1989.

G. Marsaglia, A Current View of Random Number Generators, *Computer Science and Statistics, The Interface*, Elsevier Science Publishers B. V. (North Holland) L. Billard (ed.), 1985.

W. R. Martin, P. F. Nowak and J. A. Rathkopf. "Monte Carlo photon tracing on a vector supercomputer," *IBM Journal of Research and Development 30(2)*, 1986.

M. Mascagni, S. Cuccaro, D. Pryor, and M. Robinson, "A fast, high quality, reproducible, parallel, lagged-Fibonacci pseudorandom number generator," SRC-TR-94-115, Supercomputing Research Center, 17100 Science Drive, Bowie, MD 20715, 1994.

J. D. McDonald and D. Baganoff. "Vectorization of a particle simulation method for hypersonic rarefied flow," AIAA-88-2735, AIAA Thermophysics, Plasmadynamics and Lasers Conference, San Antonio, June 27-29, 1988.

N. Metropolis, "The beginning of the Monte Carlo method," *Los Alamos Science, 15*, 1987.

N. Metropolis. "Monte Carlo - in the beginning and some great expectations," Monte Carlo Methods and Applications in Neutronics, Photonics and Statistical Physics, Cadarache Castle, France, 1985.

M. F. Modest. "Monte Carlo method in analysis of three-dimensional radiative exchange factors for non-gray non-diffuse surfaces," *Numerical Heat Transfer, 1*, 403, 1978.

M. E. Mueller. "Some continuous Monte Carlo methods for the Dirichlet problem," *Annals of Mathematical Statistics, 27*, 569-589, 1956.

S. K. Park and K. W. Miller, "Random Number Generators: Good Ones are Hard to Find," *Transactions of the ACM*, Nov. 1988.

L. C. Polgar and J. R. Howell. "Directional thermal-radiative properties of conical cavities," NASA TN D-2904, 1965

W. H. Press et al. *Numerical Recipes (FORTRAN)*, pps. 191-225, 1988.

D. V. Pryor and P. J. Burns. "A parallel Monte Carlo model for radiative heat transfer," Presented at the 1986 SIAM Meeting, Boston, MA, July 21-25,1986.

D. V. Pryor and P. J. Burns. "Vectorized molecular aerodynamics simulation of the Rayleigh problem," *Proceedings, Supercomputing '88*, Orlando, FL, Nov. 14, 1988.

D. Pryor, S. Cuccaro, M. Mascagni, and M. Robinson. "Implementation and usage of a portable and reproducible parallel pseudorandom number generator," SRC-TR-94-116, Supercomputing Research Center, 17100 Science Drive, Bowie, MD 20715, 1994.

Lord Rayleigh, "On James Bernoulli's Theorem in Probabilities," *Philosophical Magazine, 47*, 1899, pp. 246-251.

S, M. Ross. *A First Course in Probability*, 2nd Ed., Macmillan, New York, NY (1976).

R. Y. Rubenstein. *Simulation and the Monte Carlo Method*, Wiley, New York, NY, 1981.

Y. A. Schreider. *Methods of Statistical Testing*, Elsevier, New York, NY, 1964.

Sequent Computer Systems. "Parallel ray tracing study," TN-85-09(rvp), Rev. 1.0, 1985.

N. Shamsunder, E. M. Sparrow and R. P. Heinisch. "Monte Carlo radiative solutions - effect of energy partitioning and number of rays," *International Journal of Heat and Mass Transfer 16(3)*, 690-694, 1973.

S. Ulam and N. Metropolis. "The Monte Carlo method," *Journal of American Statistical Association, 44*, 335, 1949.

S. Ulam, R. D. Richtmeyer and J. von Neumann. "Statistical methods in neutron diffusion," LAMS-551, Los Alamos National Laboratory, 1947.

Y. Q. Zhong and M. H. Kalos. "Monte Carlo transport calculations on an Ultracomputer," Ultracomputer Note #46, March 1983.

G. E. Zinsmeister and J. A. Sawyer. "A method for improving the efficiency of Monte Carlo calculation for Dirichlet problems," *Journal of Heat Transfer 96(2)*, 1057-1064, 1976.