Linear, Congruential Random Number Generators

Pat Burns

GS 510

Fall 2004

Outline

I. Linear, Congruential Generators (LCGs)

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

$$X_{n+1} = aX_n + c \text{ (mod m) (1)} \qquad (1)$$

where

X_n	= n^{th} *random integer*
X_{n+1}	= $n+1^{th}$ *random integer*
a	= *the integer multiplier*
c	= *the integer additive constant*
m	= *the modulus for modulo arithmetic*

2. The next random integer X_{n+1} is generated using the previous random integer X_n, the integer constants a and c, and the integer modulus m
3. 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}
4. The method requires an initial seed, X_0, to get started
5. The initial seed, X_0, may be
   a) “Randomly” generated, generally by using the date and time functions (more later)
   b) Be set to a constant, default, internal value, or
   c) Be set to a user-specified value (input)

6. The entire sequence is characterized by the multiplier, *a*; the additive constant, *c*; the modulus, *m*; and the initial seed X_0.
7. Following Anderson [3], 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.
8. The appearance of randomness is provided by performing modulo arithmetic or remaindering.
   a) For example, the nonnegative integers 0; 1; 2; 3; 4; 5; ... modulo 3 are 0; 1; 2; 0; 1; 2; ...
   b) Note that the next result, X_{n+1}, depends upon only the previous integer, X_n - characteristic of linear, congruential generators which minimizes storage requirements, but at the same time, imposes restrictions on the period
9. The case $c \neq 0$ is termed a mixed congruential random number generator
10. The case c = 0 is termed a multiplicative congruential random number generator
11. With X_n determined, we generate a corresponding real number, R_n, as follows:

$R_n = X_n/\text{float}(m)$, with R_n on [0,1), or (2)

$R_n = X_n/\text{float}(m - 1)$, with R_n on [0,1] (3)

a) We desire uniformity, where any particular R_n is just as likely to appear as any other R_n, and

(1) The average of the R_n should be 0.5

(2) The standard deviation should be 1/12 = 0.83333...

II. Example 1 - LCG (5, 1, 16, 1)

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,...) (4)

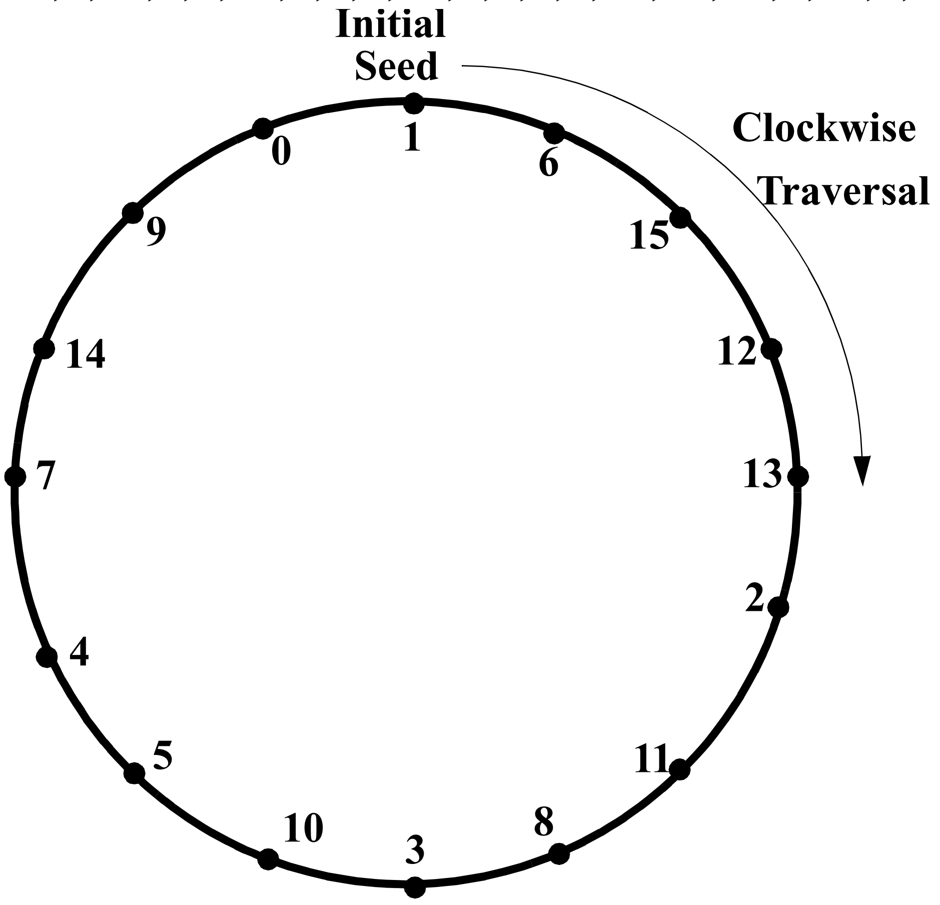


Figure 2 - The Random Number Cycle for Example 1 LCG(5,1,16,1)

2. In Figure 2, we illustrate the random number cycle for this generator. We immediately observe four features:
   a) 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.

b) 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;... Due to this lack of randomness, the X_n values should not be used as random digits (especially the rightmost 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.

c) 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*.

d) 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 = 0.5 and 1/12 = 0.08333... is due to our division by *m* rather than *m-1*, 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.

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

n	"Random" Integer, X_n	"Random" Binary, X_n	"Random" Real, R_n
0	1	0001	0.0625
1	6	0110	0.3750

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

n	“Random” Integer, X_n	“Random” Binary, X_n	“Random” Real, R_n
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

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

$$X_{n+1} = aX_n \pmod{m} \tag{5}$$

$$X_{n+1} = 5\,X_n \pmod{16} \tag{6}$$

1. As illustrated in Figure 3, we obtain the sequence

$$1,5,9,13,1,5,9,13,... \tag{7}$$

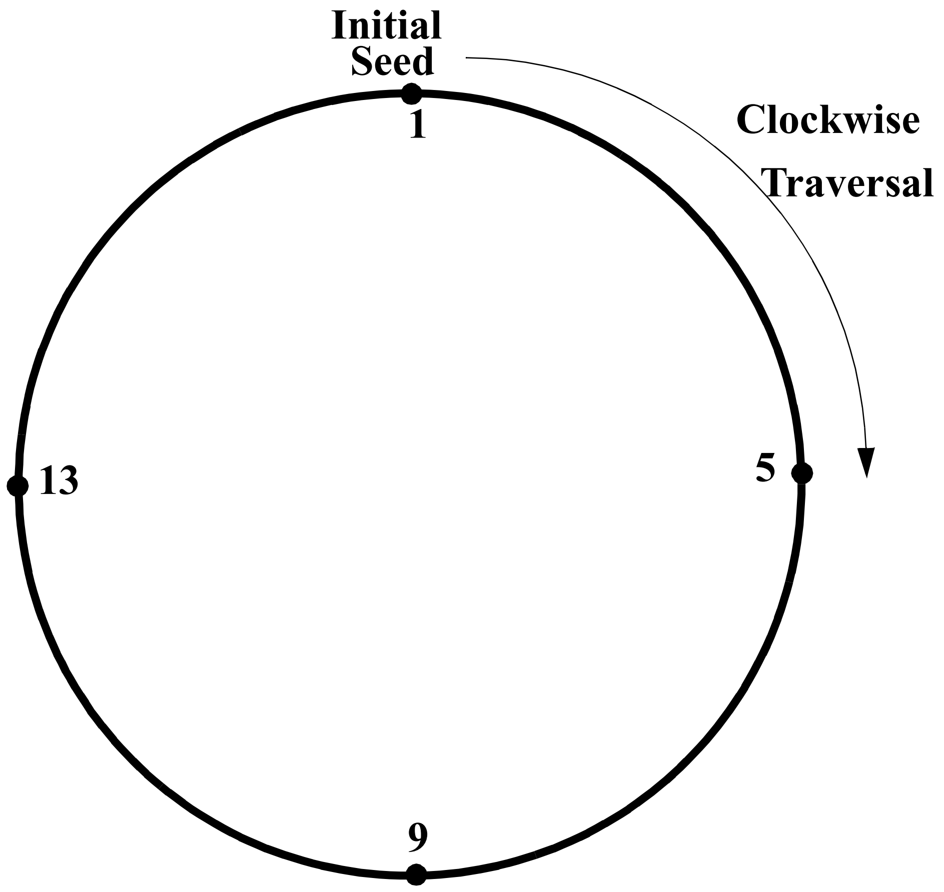


Figure 3 - The Random Number Cycle for Example 2 LCG(5,0,16,1)

2. Observations

a) 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, $m = 2^M = 2^4$) and c = 0, the maximum period is 2^{M-2}.

b) Examination of Table 2 reveals that the low order bits are not random. In fact, the two least significant bits are constant, always 01; and the most significant (leftmost) bits are quasi-random, exhibiting the pattern 00, 01, 10, and 11.

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

n	"Random" Integer, X_n	"Random" Binary, X_n	"Random" Real, R_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

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

d) 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:

(1) 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.

(2) 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

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

1. We obtain the sequence

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

2. 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).

Table 3:

n	"Random" Integer, X_n	"Random" Binary, X_n	"Random" Real, R_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	18	011100	0.7568

Table 3:

n	"Random" Integer, X_n	"Random" Binary, X_n	"Random" Real, R_n
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	0.7279
31	24	011000	0.6486
32	9	001001	0.2432
33	8	001000	0.2162
34	3	000011	0.0811
35	15	001111	0.4054
Average			0.5000
Variance			0.08333

3. Indeed, when $m = p$, a prime, the maximum period, P_{max}, is m-1, even if $c \neq 0$.
    a) Thus, for linear, congruential generators with a prime modulus, using a non-zero c does not increase the period.
4. Observations
    a) Here, the low order bits, while not exhibiting a discernible pattern, do not appear as “random” as one might expect. Indeed, as shown in Altman, the bitwise randomness 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.

V. Exercise 3. LCG (16,807; 0; 65,536; 16,811)

Obtain and edit the linear, congruential random number generator code ranlc.f. Supply the additional code to compute the mean and variance. For sample sizes, N, of N = 10, 100, 1,000, 10,000 and 100,000, run the code and plot the errors in the average and variance versus N. Discuss how the error varies with the sample size, N. Use the following parameters: a = 16,807, c = 0, m = 65,536 (=2^{16}), and X_0 = 16,811.

VI. 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 recommends setting the initial seed X_0 to the following integer:

$$X_0\text{=iyr+100*(imonth-1+12*(iday-1+31*(ihour+24*(imin+60*isec))))} \quad (9)$$

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:

$$X_0\text{=isec+60*(imin+60*(ihr+24*(iday-1+31*(imon-1+12*iyr))))} \quad (10)$$

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

$$\text{x0 = ior(x0; 1)} \quad (11)$$

VII. Overflow and Negative Integers

Consider performing the operation $aX_n + c$ (mod 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 that mask off the most significant bits (including the sign bit) to perform the modulo operation. In this case, a logical AND operation should be performed on the intermediate integer using 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.,

$$\mathrm{xn = iand(xn, imask)} \qquad (12)$$

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 [47]. Note that, in the language C, one can avoid the issue of the sign bit by using an unsigned long integer.

Figure 4 illustrates this for a 5-bit integer (in the shaded region) using the mask of 15 = 2^4-1. Note that the most significant bit (here, the 5^{th} bit from the right) is the sign bit. After the multiplication of a times X_n, a value of 730 results, exceeding the maximum 4-bit integer value of 15. After the bitwise IAND operation, the value of 9 results. Performing the modulus operation is extremely efficient, and this is the primary reason why a modulus of 2^M is used. In addition, this takes care of the issue of the sign bit.

Many computer languages have a bitwise AND intrinsic function. Use of the AND operation also avoids the problem

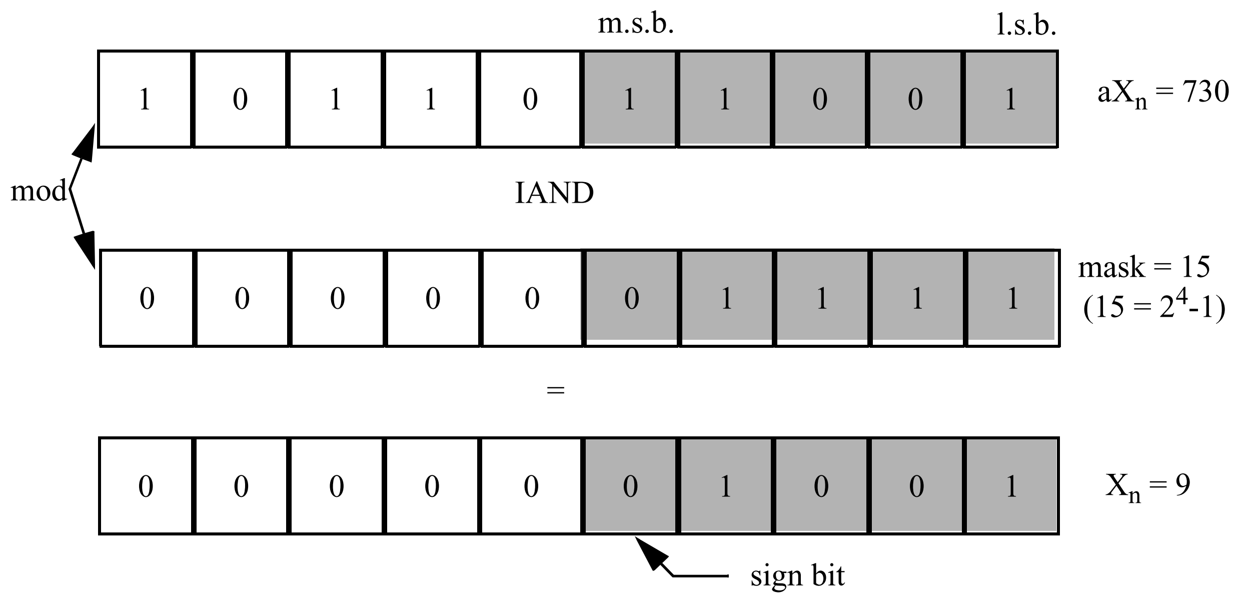


Figure 4 Modulus Operation with $m = 2^4$ (i.e. M = 4) using IAND

with negative integers, discussed above. A FORTRAN implementation would look like:

$$xn = iand(xn, mask) \tag{13}$$

VIII. Summary of LCG Properties

"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 $m = 2^M$ are faster on binary computers." - Anderson (1990)

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

   a) The full period of $m = 2^M$ is obtained if and only if $a = 1 \pmod 4$, and c is odd (often chosen as *1*).

      (1) Thus, for a full period, a must $= 4*N + 1$, where N is any integer.

   b) Low-order bits are not random!

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

   a) The maximum period of this generator is 2^{M-2} (one-quarter the modulus), and is obtained if and only if $a = 3 \pmod 8$ or $a = 5 \pmod 8$ (5 is preferred) and the initial seed is odd.

      (1) Thus, for a full period, a must $= 8*N + 3$ or $8*N + 5$ (preferred), where N is any integer.

   b) Low-order bits are not random!

3. $m = p$ (prime):

   a) The maximum period of this generator is independent of *c,* and is *p-1* and is obtained if and only if *a* is a primitive element modulo *p*.

b) Note that there is always one integer which maps only to itself. (In the case of $c = 0$, $X_0 = 0$ maps to itself.)

c) Low-order bits may or may not be random!

d) Park and Miller recommend the following, portable generator: $m = p = 2^{31}-1 = 2{,}147{,}483{,}647$; $a = 16{,}807$; and $c = 0$. The code random.f, is based upon Park and Miller's algorithm.

IX. Exercise 4 LCG Sequences

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

a) LCG(5, 0, 37, X_0)

b) LCG(5, 1, 37, X_0)

c) LCG(5, 5, 37, X_0)

Answer:

$X_{n+1} = (a*X_n+c) \bmod m$ *means:*

$(a*X_n+c) = m*N + X_{n+1}$, *where N is any integer*

For b) above

$(5*X_0+1) = 37*N + X_{n+1}$

For $N = 1$, *the answer is* $X_0 = 9$

$(5*9+1) \bmod 37 = 46 \bmod 37 = 9$

X. Exercise 5 - LCG (5, 0, 16, X_0)

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

XI. Exercise 6 - Intrinsic RNG

On a unix system, using man pages, locate a multiplicative, congruential generator i.e. $c = 0$. Read how to get and set the initial seed.

a) Get the default initial seed and discuss its impact on cycle length.

b) Set the initial seed to 1, generate the next random number, and get the new seed (the multiplier, a) again. Discuss its impact on cycle length.

c) Using the default initial seed, get and print out the first ten values in decimal, hexadecimal and real format. Comment on any observable patterns.

XII. 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.

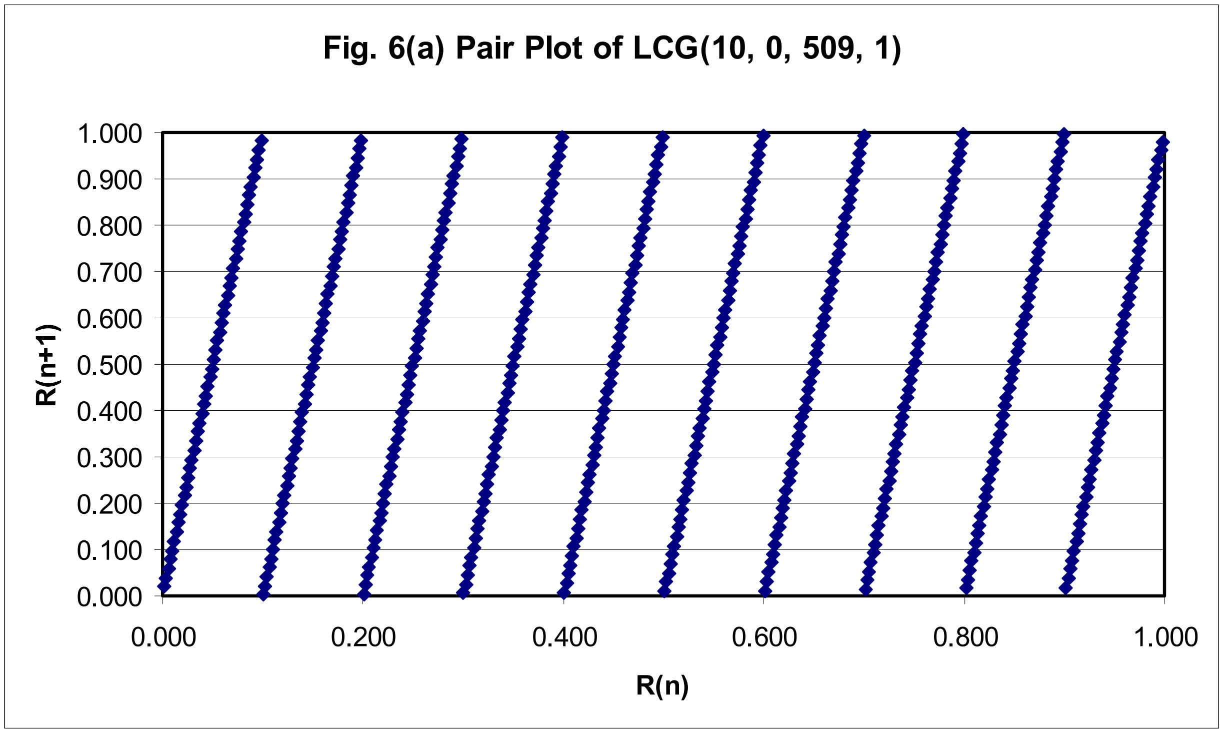
If we repeat this procedure for a large number of trials, we would like to expect that we will achieve a reasonably good "covering" of the unit square and in a "randomly" ordered fashion. We would be suspicious of a generator that produced points in the square that were, say, clustered in the bottom half, or perhaps covered the square in some clear order from left to right. Many other forms of obvious non-random 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 outputs 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.

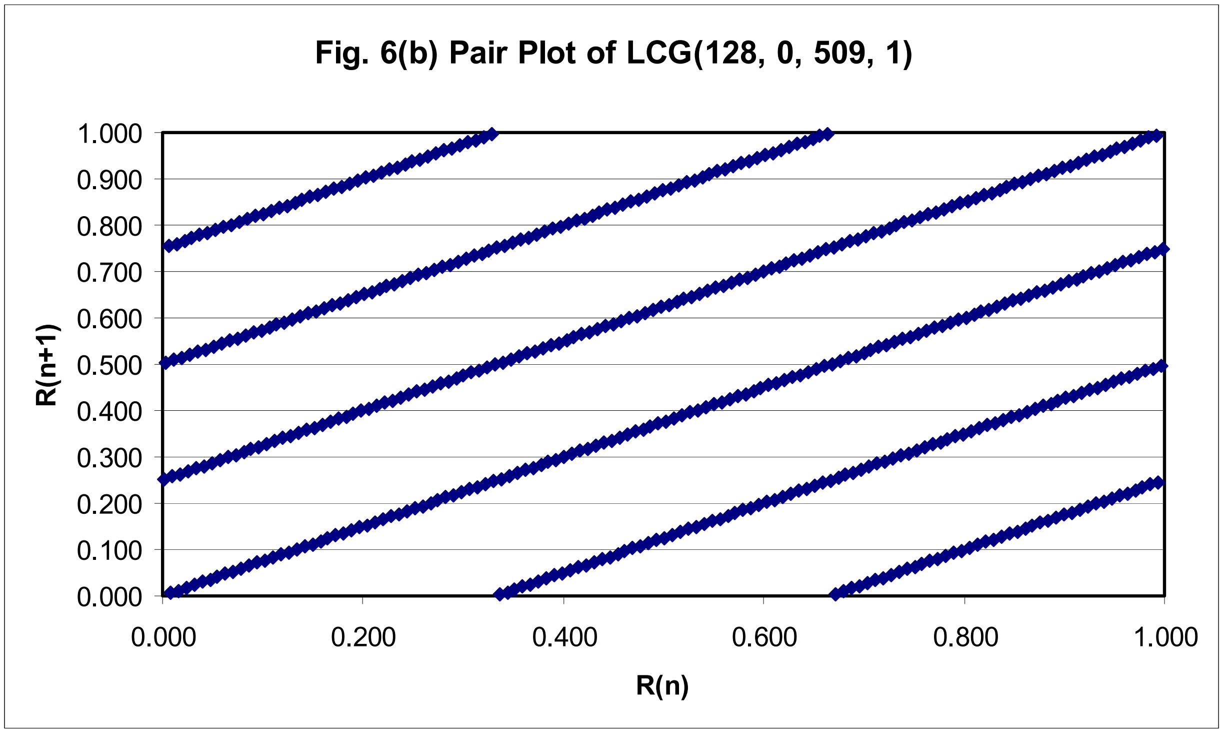
In Figures 6 a-f, we present a few examples to illustrate this behavior. These examples depict the result of generating all pairs of consecutive numbers in the period of full-period LCGs. For each modulus, the plotted points show the effect of our choice of the multiplier, a.

1. Fig. 6(a) Pair plot of LCG(10, 0, 509, 1)



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 one set of rows. The best situation is to have the maximum spacing of the rows, when viewed over all angles, as small as possible.

2. Fig 6(b) Pair plot of LCG(128, 0, 509, 1)



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."

3. Fig 6(c) Pair plot of LCG(108, 0, 509, 1)

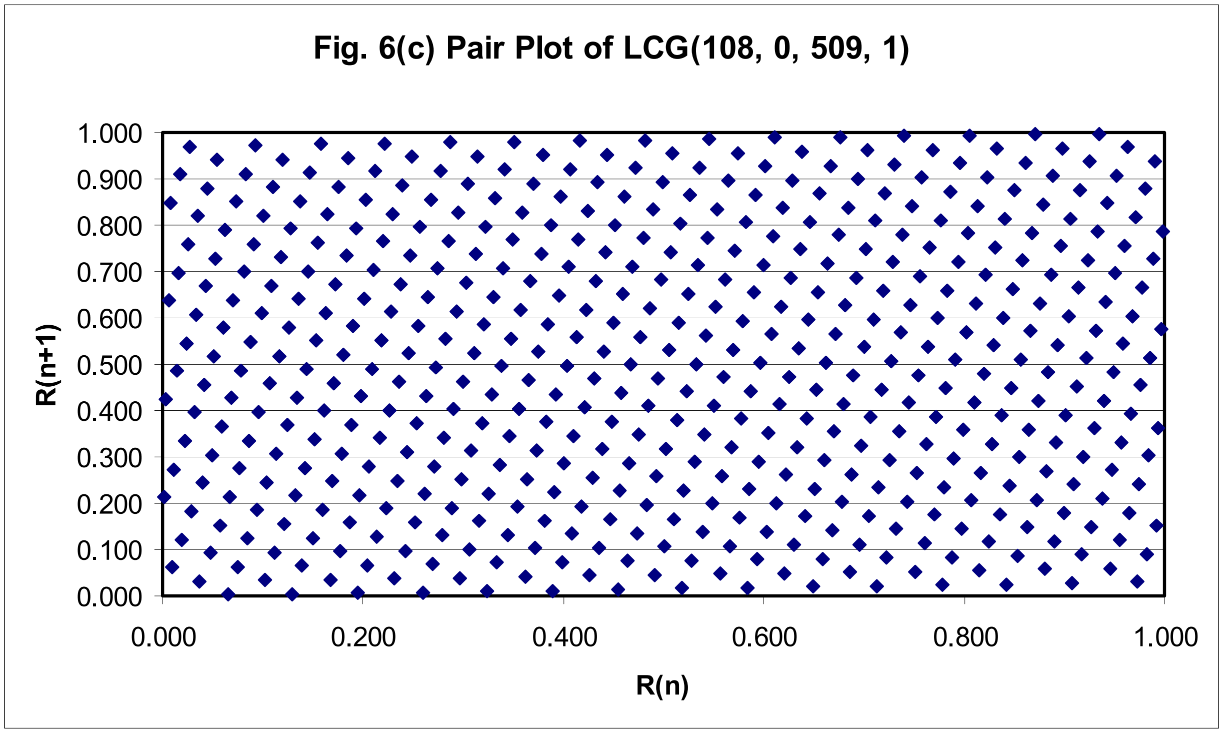


Figure 6c shows the effect of a good choice of a, LCG(108, 0, 509, 1), for which the maximum spacing between rows is clearly smaller than it is in the first two cases, and where the points cover the unit square with nearly optimal uniformity, given that with this generator we can generate only a small 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 = 3 mod 8 or a = 5 mod 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.*

4. Fig. 6(d) Paor plot of LCG(269, 0, 2048, 1)

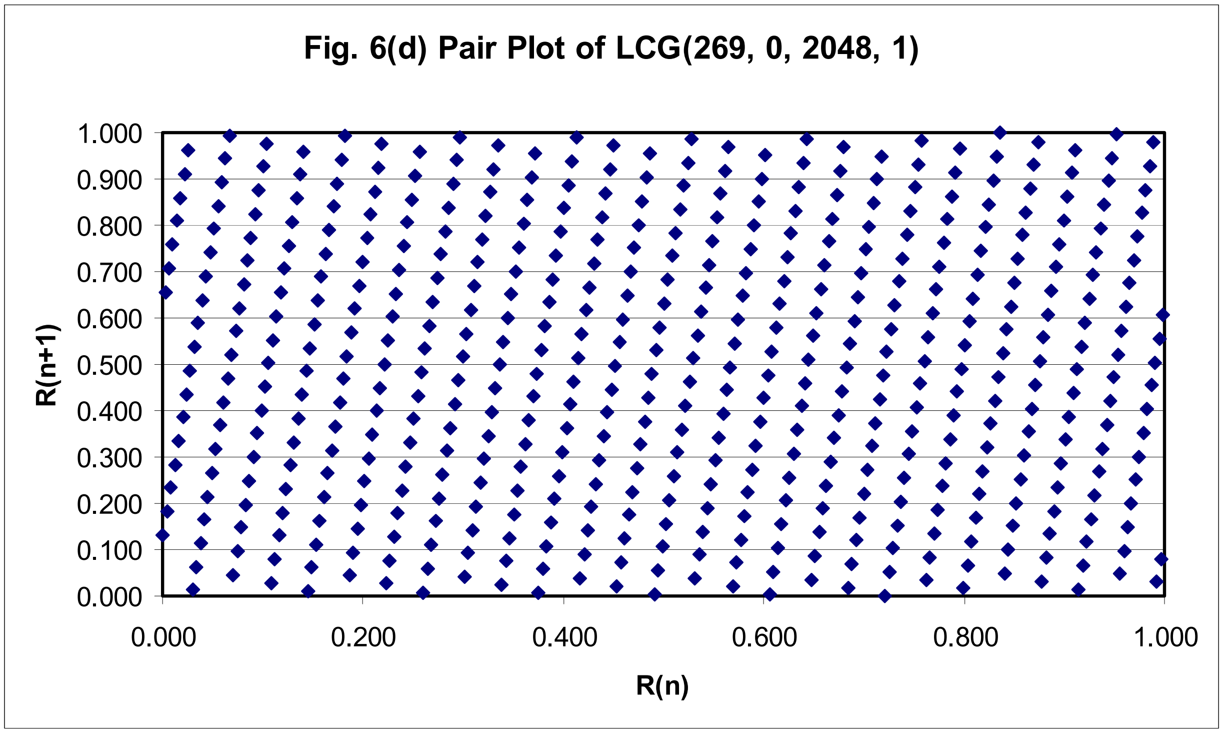
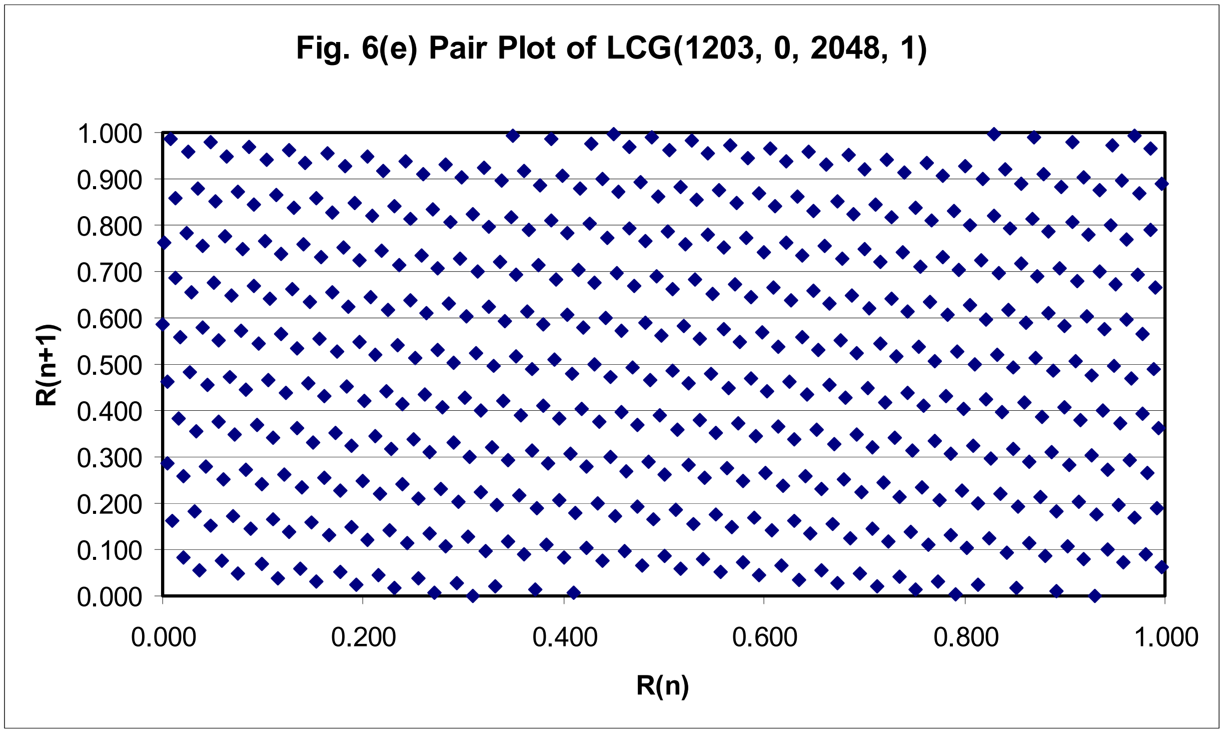


Figure 6d illustrates the result of using a good choice of a (a = 269 = 5 mod 8). As in the prime modulus case, the plotted points form a uniform looking lattice.

5. Fig 6(e) Pair plot of LCG(1203, 0, 2048, 1)



But when we plot the points in Figure 6e generated by taking a = 1203 = 3 mod 8, we see that the pattern looks like two lattice structures slightly offset from one another. This double lattice distinguishes the a = 3 mod 8 case from the a = 5 mod 8 case and is the reason that a = 5 mod 8 is preferred, even though both sets of a values produce the same period. As an aside, the two lattice structures in the a = 3 mod 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.*

6. Fig. 6(f) Pair plot of LCG(16807, 0, 2147483647, 1)

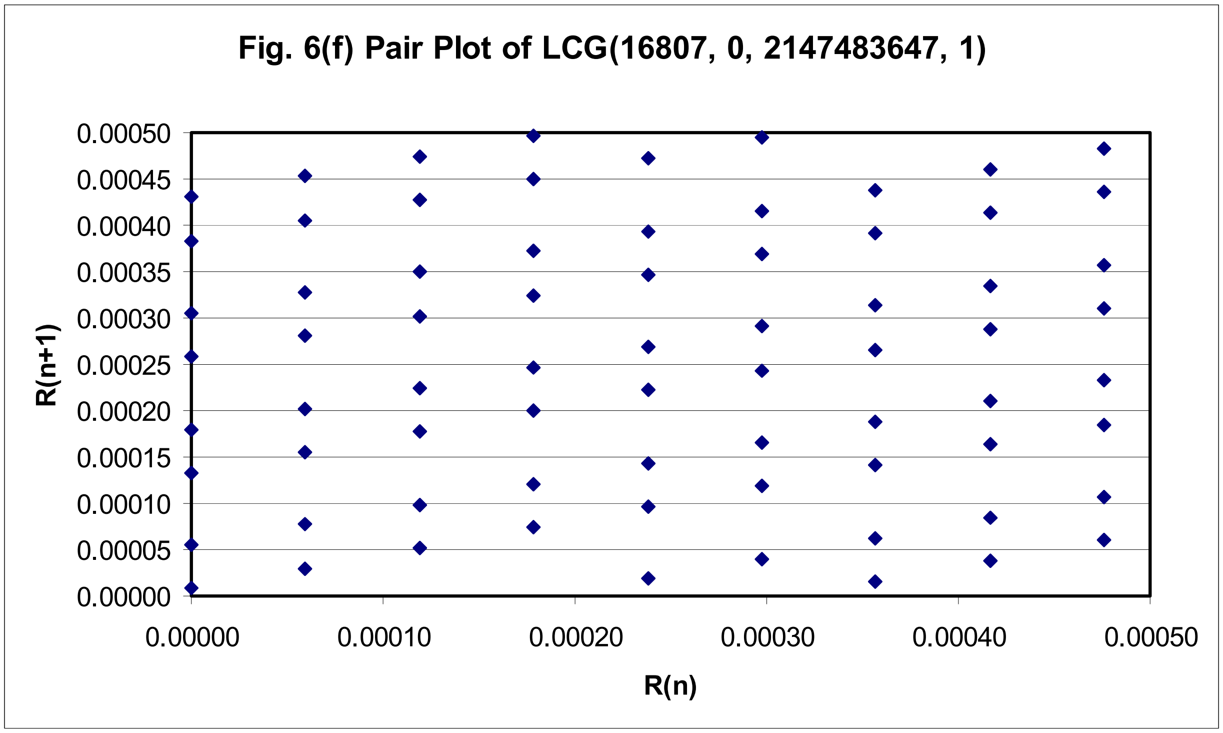


Figure 6f shows the full-period 2-D result for LCG(16807, 0, 2^{31}-1, 1), = LCG(16807, 0, 2147483647, 1), where we "zoom in" on only the portion of the unit square in the (0,0.0005) X (0,0.0005) corner, or 1/4,000,000 of the total area. Although the row spacing for this generator is obviously not optimal, 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 $a = 3 \bmod 8$; 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.*

XIII. Exercise 7 Visualization of some poor LCGs

Modify the ranlc.f or ranlc.c code to output pairs of random numbers as (R_1, R_2), (R_2, R_3), (R_3, R_4), *etc.Plot the discrete points for:*

a) LCG(66, 0, 2^{31}, 1), 4,095 points (4,096 random numbers)

b) LCG(5, 0, 2^{31}-1, 1), 4,095 points (4,096 random numbers)

XIV. Exercise 8 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 $X_{n+k} = \hat{a}_k X + \hat{c}_k$, resulting in a leap-ahead of k places in the original LCG cycle.

XV. Vectorization and Access via Multiple Processors

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 non-overlapping 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:

$$X_{n+1} = aX_n \ (\text{mod } m); \tag{14}$$

$$X_{n+2} = aX_{n+1} = a^2X_n \ (\text{mod } m) \tag{15}$$

$$X_{n+3} = aX_{n+2} = a^3X_n \ (\text{mod } m); \tag{16}$$

$$\ldots$$

$$X_{n+k} = aX_{n+k-1} = a^kX_n \ (\text{mod } m); \tag{17}$$

Thus we can "leap ahead" k places of the period by multiplying the current seed value by a^k mod 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, d, the power-of-two powers of a:

$$d_0 = a \qquad (18)$$

$$d_1 = d_0^{\,2} \qquad (19)$$

$$d_2 = d_1^{\,2} \qquad (20)$$

$$\ldots$$

$$d_L = d_{L-1}^2 \qquad (21)$$

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

$$\_d = (a, a^2, a^4, a^8, a^{16}, a^{32}, a^{64}) \qquad (22)$$

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

$$a^{91} = a^{64}\,x\,a^{16}\,x\,a^8\,x\,a^2\,x\,a^1 = d_6\,x\,d_4\,x\,d_3\,x\,d_1\,x\,d_0 \qquad (23)$$

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:

$$\text{Choose seed1} \tag{24}$$

$$\text{seed}_2 = a^k \text{seed}_1 \pmod{m} \tag{25}$$

$$\text{seed}_3 = a^k \text{seed}_2 \pmod{m} \tag{26}$$

$$\ldots$$

$$\text{seed}_N = a^k \text{seed}_{N-1} \pmod{m} \tag{27}$$

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:* $s_i = \sum_{i=0}^{j} a^i$ *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. 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:*

$$L(X) = X_L = a_L X + c_L \pmod{m} \tag{28}$$

$$R(X) = X_R = a_R X + c_R \pmod{m} \tag{29}$$

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, ... , 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.*

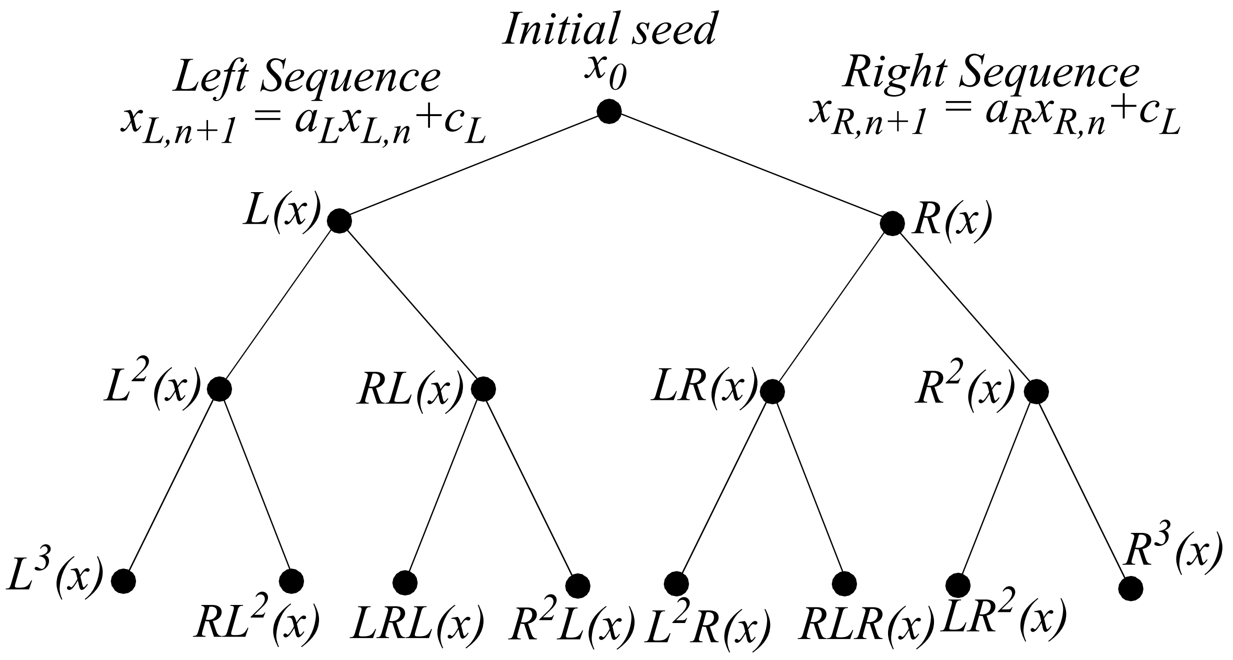


Figure 7 Tree Generated from "Left" and "Right" Generators

This is a special case of the Lehmer tree, where the left generator is used to seed a number of right generators, each typically for a separate process. As an example, consider np processes (see Foster).

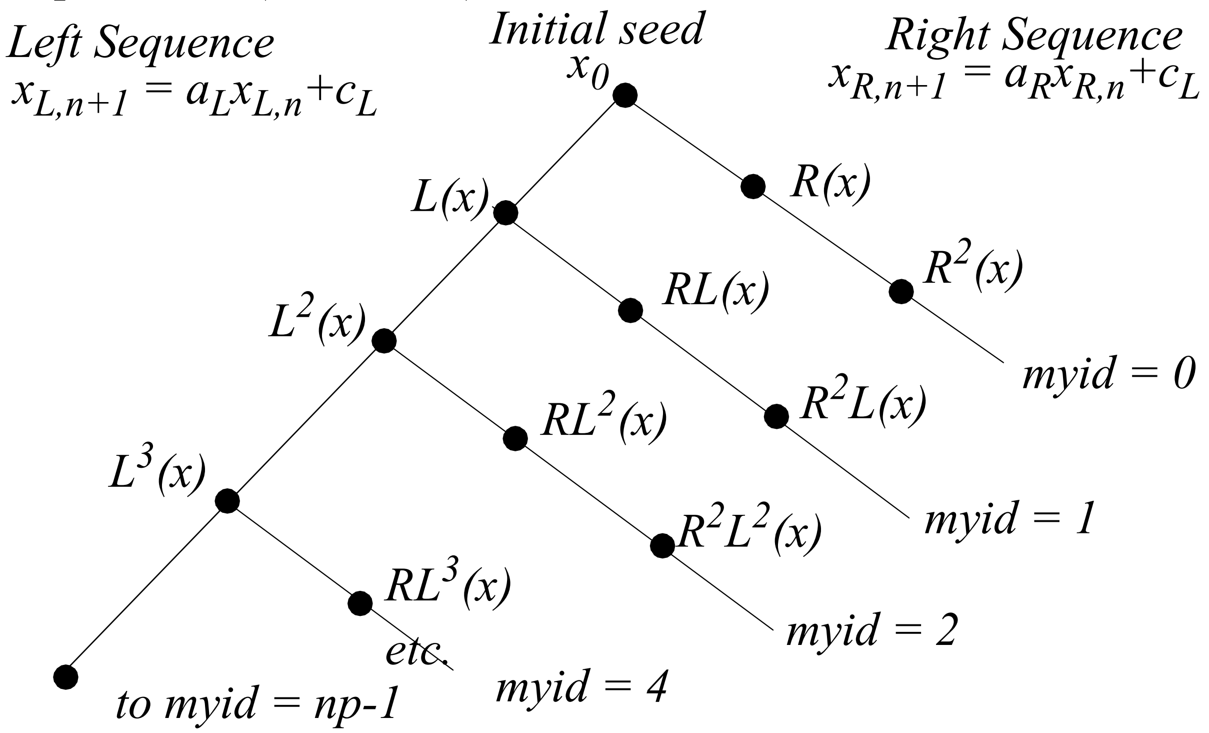


Figure 8 Lehmer Tree to Produce np "Right" Generators