Gerald P. Dwyer, Jr.; Williams, K. B.

Working Paper

Portable random number generators

Working Paper, No. 99-14

Provided in Cooperation with:
Federal Reserve Bank of Atlanta

Suggested Citation: Gerald P. Dwyer, Jr.; Williams, K. B. (1999) : Portable random number generators, Working Paper, No. 99-14, Federal Reserve Bank of Atlanta, Atlanta, GA

This Version is available at:
https://hdl.handle.net/10419/100782

FEDERAL
RESERVE
BANK
of ATLANTA

Portable Random Number Generators

Gerald P. Dwyer Jr. and K.B. Williams

Working Paper 99-14
October 1999

Working Paper Series

Portable Random Number Generators

Gerald P. Dwyer Jr. and K.B. Williams

Federal Reserve Bank of Atlanta
Working Paper 99-14
October 1999

Abstract: Computers are deterministic devices, and a computer-generated random number is a contradiction in terms. As a result, computer-generated pseudorandom numbers are fraught with peril for the unwary. We summarize much that is known about the most well-known pseudorandom number generators: congruential generators. We also provide machine-independent programs to implement the generators in any language that has 32-bit signed integers—for example C, C^{++}, and FORTRAN. Based on an extensive search, we provide parameter values better than those previously available.

JEL classification: C15

Key words: random number generators, congruential generators, Monte Carlo

---

An earlier version of this paper was presented at the Society for Computation Economics meeting in 1997, and the authors thank the participants for helpful comments. They also thank Stephen L. Moshier, who kindly provided many underlying routines used in the programs and helpful comments. The computer code and detailed tables are available from either author. The views expressed here are the authors' and not necessarily those of the Federal Reserve Bank of Atlanta or the Federal Reserve System. Any remaining errors are the authors' responsibility.

Please address questions regarding content to Gerald P. Dwyer Jr., Research Department, Federal Reserve Bank of Atlanta, 104 Marietta Street, NW, Atlanta, Georgia 30303-2713, 404/614-7095, 404/521-8810 (fax), gerald.p. dwyer@atl.frb.org (or dwyerg@clemson.edu); or K.B. Williams, 412 Magnolia Avenue, Melbourne Beach, Florida 32951-2000, kbwms@aol.com.

I. INTRODUCTION

Economists use computer-generated random numbers in applications that range from commonplace ones simulation to relatively novel ones optimization and estimation. Despite their ubiquity, the generators used in such applications are mentioned seldom and if so, only elliptically. This is unfortunate because the generator used is an extremely important part of any study that uses a random number generator. In some ways, not mentioning a generator used is equivalent to presenting estimated equations and not mentioning the data sources or the estimator. As we indicate below, there are good reasons to be dubious about some generators readily available.

The beginning of wisdom about computer-generated random numbers is to remember always: computer-generated random numbers are not random. Indeed, computer-generated random numbers are called pseudorandom numbers in the literature and we follow that practice.[1] Commonly available pseudorandom numbers are generated by deterministic nonlinear difference equations. Hence, in the sense of being independent across observations, pseudorandom numbers definitely are not random. Furthermore, computers have finite memory. Hence, computers can only approximate real numbers from a continuous distribution. In addition, computers are finite-state machines. Hence, any sequence of numbers must eventually repeat because the computer-represented nonlinear difference equation eventually will return to an earlier state.

Given these failures to represent the underlying mathematical construct, how can pseudorandom numbers be useful at all? While it is quite possible to get into an extended discussion about the meaning of the term random, it is more informative to note that the typical reason for generating pseudorandom numbers in economics and finance is to generate numbers that *appear* to be generated by a distribution

---

[1] There also are quasirandom number sequences, which are well-behaved in some uses and have advantages compared to known pseudorandom number sequences but serious defects for other uses (Niederreiter 1992).

function, most commonly the uniform distribution.[2] If pseudorandom numbers behave in their intended use as if they were generated by a uniform distribution or some other desired distribution, that is all that need be asked. Furthermore, the inability of a computer to represent the continuum is not unique to random-number generation.[3] Finally, the finite length of any sequence of pseudorandom numbers only implies that a user could use a long enough sequence of numbers that the generator wraps around to the beginning.[4]

In this article, we provide background helpful for understanding some alternative random number generators and recommend a particular generator that can be used in almost all, if not all, environments. This discussion is a mix of general mathematical considerations and some general programming considerations.[5] We do not recommend blind acceptance of generators. Still, if you are willing to take our word for the quality of the generators, the Appendix provides computer code for faster and better generators than those suggested by Press *et al*. (1992, pp. 274-85), Knuth (1998) and others.

II. CHOOSING AMONG GENERATORS

There are a large number of different classes of pseudorandom number generators available. Knuth (1998, Ch. 3), James (1990) and Niederreiter (1992, Chs. 7-10) survey many types of generators.

Congruential Generators

---

[2] Pseudorandom numbers from some other distribution, for example the normal, are generated by a transformation of the uniformly distributed numbers.

[3] For that matter, the inability to generate the continuum is not unique to computers.

[4] The entire sequence inevitably has properties inconsistent with independent, uniformly distributed numbers. It is undesirable to use such a large part of the entire sequence that these properties may determine properties of the answer to the question being asked.

[5] In earlier work, we (Dwyer 1995; Dwyer and Williams 1996a; Dwyer and Williams 1996b) have discussed programming considerations and the code. In this paper, we focus on the general mathematical considerations and avoid programming details.

In this paper, we focus on a particular class of generators: congruential generators. Such generators, introduced by D. H. Lehmer in 1951, are common and are the basis of many elaborations. While generators such as those proposed by Marsaglia and Zaman (1991) and related ones have received a great deal of attention in recent years, not all of this attention is complimentary (L Ecuyer 1997). The properties of congruential generators are well understood.

A *linear* congruential generator is

$$x_i = (ax_{i-1} + c) \bmod m, \tag{1}$$

where x_i is the i th member of the sequence of pseudorandom numbers, a is a multiplier, c is an additive constant, m is the nonzero modulus and the mod operator means that $(ax_{i-1} + c) \bmod m$ is the least nonnegative remainder (or the residue) from dividing $(ax_{i-1} + c)$ by m.

Many versions of linear congruential generators set the constant c to zero. The resulting *multiplicative* congruential generator is

$$x_i = ax_{i-1} \bmod m. \tag{2}$$

A related way of writing (2) is $ax_{i-1} \equiv x_i \pmod m$, a usage which is consistent with saying in English ax_{i-1} is $x_i \bmod m$. For example, eight is two mod three. [6]

It is common to discuss pseudorandom number generators in terms of nonnegative integers and we do so. At first glance, it may seem strange to characterize the values from generators by integers. Performing the arithmetic in integers, though, makes it relatively simple to be certain that computations are exact: The computations are exact if the integers computed are representable in the space provided for the integers.

---

[6] The name congruential generator comes from eight being congruent with two mod three. Giblin (1993) is a useful background reference on the number theory relevant for congruential generators.

The transformation of integers to decimal numbers is straightforward if done with care.[7] The resolution of the resulting decimal pseudorandom numbers is the distance between adjacent decimal values normalized to the unit interval and this distance is limited by the inverse of the modulus. If the modulus is, for example, 2^{31}-1, this limiting distance is about $4.7 \cdot 10^{-10}$.

By combining congruential generators, it is possible to substantially improve the performance of these generators.[8] While it is possible to combine more than two generators, it is adequate for many purposes to combine just two multiplicative generators. The underlying generators of sequences $\{y_i\}$ and $\{z_i\}$ are

$$y_i = a_y y_{i-1} \bmod m, \qquad z_i = a_z z_{i-1} \bmod m. \tag{3}$$

Without loss of generality, suppose that $m_y > m_z$. These sequences could be added or subtracted, but it is easier to avoid overflow by subtracting them and that is what existing programs do, so we use subtraction. The combined generator of the sequence $\{x_i\}$ is

$$x_i = (y_i - z_i) \bmod m_y. \tag{4}$$

The final mod operation on the difference keeps the sequence of pseudorandom numbers on $[1, m_y - 1]$.

---

[7] There are pitfalls to be avoided in transforming to decimal numbers (Monahan 1985). For example, many uses of pseudorandom numbers fail for a value of zero and some for one, so it is common to restrict the range to (0,1) rather than [0,1]. If m is odd, dividing pseudorandom numbers on $[1, m-1]$ by m generates a uniform distribution on $[1/m, (m-1)/m]$. The mean of a uniform distribution is 0.5, which is accomplished by this division with m odd and more generally can be accomplished by keeping the range equidistant from 0.5 on the left and right.

[8] A higher-order difference equation is another generalization of multiplicative generators that can be used instead of a combination generator or in association with a combination generator. L Ecuyer (1996) discusses higher-order difference equations, which we do not pursue.

The combination generator (4) is a powerful generalization of the multiplicative generator. L Ecuyer and Tezuka (1991) show that the generator (4) is approximately equivalent in pertinent respects to a multiplicative congruential generator with multiplier and modulus given by[9]

$$
\begin{aligned}
m &= m_y m_z \\
a &= \left(a_y n_y m_z + a_z n_z m_y\right) \bmod m,
\end{aligned}
\tag{5}
$$

where

$$
\begin{aligned}
n_y &= m_z^{m_y - 2} \bmod m_y \\
n_z &= m_y^{m_z - 2} \bmod m_z.
\end{aligned}
\tag{6}
$$

This implies that, if m_y is 2^{31}-1 and m_z is 2^{31}-19, each about $2.15 \cdot 10^9$, the combination generator is approximately equivalent to a generator with a modulus of about $4.61 \cdot 10^{18}$.

Criteria for Choosing Among Generators

Making choices among generators requires criteria. We use the following criteria for comparing generators: consistency of the computer program s final output with the underlying mathematical model; portability of the computer code; length of the generator s period; and speed.

The desirability of having the final output consistent with the mathematical model is obvious, but achieving this goal depends on the application. It is possible to examine pseudorandom number generators to determine their consistency with the intended uniform distribution and we do this below. Such tests,

[9] The equivalence is exact for the slightly different combined generator introduced by Wickman and Hill (1982).

however, are useful but not sufficient. As happened in physics applications (Ferrenberg and Landau 1992; Schmid and Wilding 1995), it is possible for a generator to be fine in some applications and to produce incorrect results in others.[10] One way to test a generator in a particular application is to use it in a simpler version of the application with a known answer to obtain evidence of the generator s adequacy in this application.

Portability of computer code across machines is quite important. If code is not portable, it is quite possible that a program will produce different results when compiled and run on different machines (for example, a Sun and a PC) if the program even compiles on all other target machines. Cleaning up such differences in output easily can take more time than ever is saved by faster execution due to tricks possible on one machine but not another. Consistent with this emphasis, the code for the generators in this paper is available in a general application language, ANSI C++, is easily translated to C and FORTRAN and can be used in programming environments such as GAUSS. All that is necessary is that 32-bit signed integers be available. For combination generators, we require that multipliers be *approximately factorable*. (Schrage 1979.) In the appendix, we explain how approximate factoring avoids overflow when a and x_{i-1} are multiplied and compares the computational speed of approximate factoring to alternative algorithms.

Because pseudorandom number sequences must eventually repeat, all generators have a finite cycle length. Everything else the same, it is preferable to have a longer cycle length, although there is a diminishing marginal value as the cycle length increases. Some early random number generators on PCs had cycle lengths of 2^{15}-2 (32,766), which is too short for all but the simplest applications. On the other hand, a random number generator recently proposed has a cycle length of about $3 \cdot 10^{171}$ (Marsaglia and Zaman 1991). This number is at best hard to contemplate, but one comparison is to the age of the universe,

[10] The generator at issue in these physics articles is a shift-register pseudorandom number generator, not the ones discussed in this article. Niederreiter (1992, Ch. 9) discusses shift-register generators in detail.

about 10 billion years or $3 \cdot 10^{23}$ microseconds (one-millionth of a second). Given any likely technology in the foreseeable future, it is not possible to generate a full sequence of these pseudorandom numbers.

The fourth criterion, speed, is a consideration because simulations can take substantial machine time. The algorithm that we recommend is reasonably quick while being portable to all computers. Because advances in technology continually make machines faster, we put relatively little emphasis on speed. Still, speed often is important. We suggest translating a portable program into Assembler tweaked for speed rather than using code that cannot be feasiblely implemented on other machines.

III. PROPERTIES OF FULL CYCLES

The period of a congruential generator is a mathematical property of the generator that can be determined analytically. In this section, we summarize theorems concerning the periods of linear, multiplicative and combined congruential generators. We also discuss the lattice structure of pseudorandom numbers from these generators and provide results from the related spectral test.

Maximal Period

For linear congruential generators such as (1), the maximum possible period of the difference equation equals the modulus, m, and the values for this period cannot range beyond the interval $[0, m-1]$.[11] Multiplicative generators have a smaller maximum possible period because the value zero, if it ever occurs, is an absorbing state ($0 \equiv 0 (\mathrm{mod}\, m)$). Hence, with the constant set to zero, the maximum possible period of the multiplicative generator is $m-1$ on the interval $[1, m-1]$.

Most combinations of values of the multiplier a and the modulus m in (1) cannot generate sequences with the maximum possible period. Knuth (1998, pp. 10-23) summarizes much that is known about the relationship between values of the multiplier and modulus and the period of the difference equation. A useful theorem for evaluating the period of linear congruential generators is

---

[11] The period is the minimum value of T such that if the first observation is dated 0, $f^T(x_0) = f(x_0)$, where $f^T(\cdot)$ denotes the composition of the function f T times.

Theorem 1 (Knuth 1998, pp. 17-19):

> The sequence x_i generated by a linear congruential generator (1) has the maximum possible period m if and only if
>
> 1. the constant c and the modulus m are relatively prime;
> 2. for every prime p that is a factor of m, p also is a factor of $a-1$;
> 3. if 4 is a factor of m, 4 is a factor of $a-1$.

Two numbers c and m are relatively prime if their greatest common divisor is 1, which can be written $\gcd(c,m)=1$. This is satisfied trivially if c or m is prime, although neither c nor m need be prime for c and m to have no common divisors.

Another useful theorem applies to the multiplicative congruential generator (2). The theorem uses the definition of a primitive root. A number a is a primitive root mod m if: 1. a and m are relatively prime; and 2. the smallest positive integer k such that $a^k \equiv 1 \pmod{m}$ is the number of integers x satisfying $x \in [1,m]$ and $\gcd(x,m)=1$.

Theorem 2 (Knuth 1998, pp. 19-21):

If the initial value of the sequence of pseudorandom numbers is relatively prime to m and a is a primitive root mod m, the period of a multiplicative congruential generator (2) is $\phi(m)$, where $\phi(m)$ is

1. for m a positive power e of 2, i.e. $m=2^e$,
   for $e=1$, $\phi(2)=1$;
   for $e=2$, $\phi(4)=2$; for e=1, $\lambda(2)=1$;
   for $e \geq 3$, $\phi(2^e)=2^{e-2}$.
2. for m a positive power of a prime, p, i.e., $m=p^e$, $e\$1$,
   $\phi(p^e)=p^{e-1}(p-1)$.
3. for m a composite of k different primes, denoted p_1 to p_k, raised to positive powers e_1 to e_k, i.e., $m = p_1^{e_1} p_2^{e_2} \ldots p_k^{e_k}$,
   $\phi(p_1^{e_1} p_2^{e_2} \ldots p_k^{e_k}) = \mathrm{lcm}[\phi(p_1^{e_1}), \phi(p_2^{e_2}), \ldots, \phi(p_k^{e_k})]$.

The notation $\mathrm{lcm}(.,.)$ denotes the least common multiple of the numbers inside the parentheses. The conditions implied by Theorem 2 on multipliers are more involved than Theorem 1 and programming

considerations become important in finding such multipliers (Knuth 1998, p. 22; Fishman 1996, pp. 592-595). Rather than pursue the general implications or the programming implications, we make a couple of remarks illustrating the implications for the most common generators. The most common moduli are powers of two and primes that fit in common languages native types of integers.

One common generator combines a modulus of 2^{32} with a nonzero constant. There are thirty-two bits in registers on many machines, for example Intel Pentiums, and the number of distinct integers representable in a register is 2^{32}. Theorem 1 implies that a generator with a modulus of 2^{32} will have the maximum possible period of about 4.29 billion if the constant c is odd and the multiplier minus one, $a-1$, is divisible by 4. If a modulus of 2^{32} is used without a constant, Theorem 2 implies that the maximum possible period of the generator is about 1.07 billion.[12] Hence, with a modulus of 2^{32}, adding a nonzero constant to a multiplicative congruential generator can increase the period from roughly 1.07 billion to 4.29 billion. A fourfold increase in cycle length is a significant return from an addition at each cycle; therefore, adding a constant to a generator with a modulus of 2^{32} is useful.

A modulus of 2^{32} is in common use, for example in Microsoft s and Borland C compilers. In part because of results obtained using the collision test that are presented below, we recommend against using these generators without a thorough analysis of the generator.

Another common modulus is 2^{31}-1, the largest signed integer representable in a register on many machines and in native types in FORTRAN and implementations of C and C^{++}. The number 2^{31}-1 is prime. With this modulus, Theorem 2 implies that the maximum possible period with a constant is 2^{31}-1 and the maximum possible period without a constant is 2^{31}-2. Adding a nonzero constant to a multiplicative congruential generator with this modulus adds one pseudorandom number to the period of about 2.15

[12] This maximum period is achieved with a cycle of odd numbers, which implies that the seed must be odd.

billion. This is scant return in exchange for even the relatively little computer time necessary to do the additions.

While a couple of billion pseudorandom numbers are adequate for some applications, they are not adequate for many applications in economics and finance. Uses of pseudorandom numbers are likely to become increasingly demanding, and indeed, one recent study of stochastic volatilities (Kim et al., 1996) uses almost a full cycle of a congruential generator. With current technology, it is not hard to run through a full cycle: It takes about seven and a half minutes on a Pentium 400 to generate the entire sequence of numbers from a full-cycle multiplicative generator with a period of 2.15 billion.[13]

A combined generator can have a dramatically longer period than either of the constituent multiplicative generators. The period of combination generators can be determined from the following theorem:

Theorem 3 (L Ecuyer 1988, p. 744).

> If the moduli m_y and m_z in the combination generator (4) are prime and each of the underlying generators in (3) has the maximal period indicated in Theorem 2, the period of the combination generator is $(m_y - 1)(m_z - 1) / 2$.

This theorem implies that the period of a combination generator that combines a generator with a modulus of 2^{31}-1 and another generator with a different machine-representable prime modulus on the order of 2^{31} can have a cycle length of about $2.31 \cdot 10^{18}$, which is dramatically larger than either individual generator s cycle length of about $2.15 \cdot 10^{9}$. To get a perspective on this, if it takes about 7.5 minutes to generate a full cycle of numbers from a multiplicative congruential generator, it takes about 15,000 years go through a full cycle of numbers from the combination generator.

---

[13] This is only a rough indication of timing. The program was run under Windows NT with other non-computer-time-intensive operations being performed and some printing to the screen during execution.

The Lattice Structure of Congruential Generators

No matter how long or short their periods, congruential generators are deterministic difference equations. These equations have some of the same characteristics as linear difference equations and phase diagrams are an informative way to examine the behavior implied by the equations.

A phase diagram for a complete sequence of numbers generated by a uniform distribution on a discrete grid is simple: each point is equally likely, and the consistency with a uniform distribution is indicated by the evenness with which the numbers are spaced out. For example, if 13-bit numbers are used, there are $2^{13}=8192$ different values. There are 8192^2 points in a phase diagram showing the relationship between x_{i+1} and x_i because the next value can be any of the 8192 different values no matter what the prior values and there are 8192 different possible values on [0, 8191].

It is informative to compare a uniform distribution of integers on $[0,8191]^2$ to the output of a multiplicative congruential generator such as

(7) $$x_i = ax_{i-1} \bmod 8191.$$

The modulus 8191 is the largest prime number less than 2^{13}. If the multiplier a is a primitive root of 8191, the period of this generator is 8190. It is obvious that equation (7) can generate only one value of x_i from any particular value of x_{i-1} on [1, 8190]. In addition, if the multiplier in (7) generates the maximal period, x_i visits every integer from 1 to 8190 in a full cycle. Hence, the generator must associate each value of x_i from 1 to 8190 with one and only one value of x_{i-1} and equation (7) can visit only 8190 of the 8192^2 two-dimensional points representable by 13 bits. In higher dimensions, the fractions of points visited are smaller and smaller fractions of all of the points. This failure to visit all the points is a property of any one-to-one deterministic difference equation, so this failure cannot be used to dismiss congruential generators relative to other generators.

A great deal is known about the phase diagram for congruential generators. We illustrate the issues with the modulus of 8191 and two multipliers: 2066 and 2341. Both multipliers are primitive roots of 8191 and generate the maximum cycle length of 8190 for a multiplicative generator with a modulus of 8191.

Figure 1 shows the two-dimensional phase diagrams and clearly indicates one important difference between these two multipliers: the spacing of pseudorandom numbers. The pseudorandom numbers for the multiplier 2066 in the graph in the left-hand side of Figure 1 are relatively uniformly spread around the graph. With the same modulus of 8191, the multiplier 2341 produces the set of pseudorandom numbers on the right-hand side of Figure 1. With a multiplier of 2341, all of the pairs of numbers can be characterized as lying on eleven parallel straight lines. Parallel lines also can be drawn for the multiplier of 2066, but more lines are necessary to include all the points, the distances between adjacent lines are less and there are smaller holes areas with no points in the graph.

Figure 2 shows three-dimensional phase diagrams for these multipliers. The set of hyperplanes is less apparent than in Figure 1, but the structure is there. In two, three and higher dimensions, as this figure suggests, the phase diagrams continue to have the lattice structure. The three-dimensional graphs again suggest that the multiplier of 2341 is not as good at approximating a uniform distribution as is a multiplier of 2066. With a multiplier of 2341, the minimum number of planes to cover the points is fewer, these planes are farther apart, and holes appear to be larger. Overall, the multiplier of 2341 provides a substantially worse approximation to a uniform distribution.

The Spectral Test

These insights from viewing the graphs are formalized in what is known as the spectral test for congruential generators (Knuth 1998, pp. 93-118; Fishman 1996, pp. 611-28.)[14] As in the prior illustration,

[14] The name spectral test is used because this insight about the structure of congruential generators was instigated by a proposed test of congruential generators using a frequency representation.

pseudorandom numbers from a linear congruential generator always can be covered by a set of equidistant parallel hyperplanes.

The spectral test computes the maximum Euclidian distance between adjacent hyperplanes among the set of hyperplanes that cover all the points. Rotation of the set of hyperplanes relative to the axes does not affect the measure. Because of this and because a constant in a congruential generator merely rotates the points in the phase space, spectral test results are independent of any value of the constant term. The maximum distance for the d-dimensional phase space can be written

(8) $$d^*(a,m) = \max_{q_j} d(a,m,q_j)$$

where q_j denotes the j th possible vector of parameters characterizing the hyperplanes and $d(a,m,q_j)$ is the Euclidian distance between the hyperplanes generated by the vectors q_j. For congruential generators, the vectors q_j necessarily consist of integers. Compared to other possible measures such as the average distance between hyperplanes, the maximum distance is a conservative measure because it focuses on the largest distance across the dimensions analyzed.

This distance is quite valuable for comparing different multipliers for a given modulus. For example, the spectral test results can be used to compare the multipliers 2066 and 2341 in Figures 1 and 2. This distance is less informative for comparing generators with large differences in the moduli though, because the distance between adjacent values, resolution, is an additional criteria for distinguishing between moduli. The resolution of a full-period generator depends on the multiplier because the resolution is $1/m$. However, for relatively small differences between moduli such as 2^{31}-1 and 2^{31}-19, the difference in resolution is unimportant.

A normalization of the maximum distance between hyperplanes makes it easier to interpret the spectral test values. No matter how the hyperplanes are rotated, the minimum possible values of the maximum distance between hyperplanes are known (Cassels 1971). These known minimums are generated with real numbers possible as the parameters characterizing the lattice and cannot be improved by a

congruential generator because a congruential generator restricts the parameters to integers. Let $\delta_d(m)$ denote the minimum possible maximum distance between equidistant parallel hyperplanes in d dimensions. The normalized selection criterion for comparing the distance between hyperplanes for a given modulus is

$$S_d(a,m) = \frac{\delta_d(m)}{d^*(a,m)} \text{ with } S_d(a,m) \in (0,1]. \tag{9}$$

A value of one would indicate that the multiplier achieved the minimum possible maximum distance. The value of $S_d(a,m)$ goes to zero as the distance between adjacent hyperplanes increases.[15]

The spectral test results for the multipliers of 2066 and 2034 with a modulus of 8191 are consistent with the inferences from a visual inspection of Figures 1 and 2. The spectral test values for two and three dimensions are 0.75 and 0.76 for the visually better multiplier of 2066 and 0.09 and 0.38 for the visually worse multiplier of 2341.[16]

Spectral Test Results

We have run spectral tests to determine good multipliers for the combined generator. With current computation power, it is not particularly difficult to run exhaustive tests for multiplicative generators and

[15] The spectral test would not be feasible if the test required a complete set of all pseudorandom numbers and searching for the maximum distance between sets of parallel hyperplanes by complete enumeration. Instead, using the method suggested by Dieter (1975) and explained by Knuth (~~1981~~1998, pp. 101-104), substantial simplifications of the computations are possible. L Ecuyer and Couture (1996) discuss more recent improvements of the algorithm.

[16] The spectral test values on the order of 0.75 appear good relative to the values for moduli on the order of 2^{31} but these generators have very different resolutions: the space between points is on the order of 10^{-4} for the modulus of 8191 and on the order of 10^{-10} for a modulus of 2^{31}-1. Spectral test results normalized by the minimum possible maximum distance can be compared without reference to resolution for moduli that are similar in magnitude.

Fishman and Moore (1992) did so. A requirement that multipliers be approximately factorable limits the multipliers that can be considered.[17]

There are too many possible combination multipliers to consider for us to run exhaustive spectral tests given available computational power. On a Pentium 400, it is possible to run approximately six million spectral tests on combination multipliers in a day. The number of combinations, however, is far greater than this. For example, the modulus 2^{31}-1 is prime and the next smaller prime number is 2^{31}-19. The modulus 2^{31}-1 has 23,093 multipliers that are both primitive roots mod 2^{31}-1 and approximately factorable; the modulus 2^{31}-19 has 30,873 multipliers that are both primitive roots mod 2^{31}-19 and approximately factorable. This implies that there 712,950,189 combination multipliers that can be considered. It would take about 119 days to do an exhaustive search for these two moduli. Worse, there is no reason to confine a search to these moduli. We examine combination multipliers that have moduli for the underlying generators that are the seven largest prime numbers less than 2^{31}.

It might seem that a directed search would be useful, but preliminary analysis indicates that there is no regularity in the values from the spectral test. Figure 3 shows the surface for spectral test values based on an evenly-spaced selection of multipliers for the moduli 2^{31}-1 and 2^{31}-19. Examination of the surface from alternative viewpoints uniformly leads to the same conclusion: there is no obvious regularity in the surface to direct a search.

As a result, given two moduli, we perform a *random* search over multipliers that might be used in a combined generator.[18] We use the multiplicative generator in Dwyer (1996) to direct the choices of

---

[17] With a modulus of 2^{31}-1, the best multiplier is 742038285. This generator can be efficiently implemented using Hörmann and Derflinger s (1993) version of Payne *et al*. s (1969) algorithm. Subject to the constraint of using approximate factoring, the spectral test in eight dimensions indicates that the best five multipliers with a modulus of 2^{32}-1 are 45991, 37857, 33640, 32397, and 32022.

[18] To simplify the computations, we pick a multiplier for one modulus and then do 1000 random choices of multipliers for the second modulus. We have no reason to think that picking both multipliers at random would have improved the results and it would have slowed down the computations.

multipliers. Table 1 shows the moduli examined and the number of times that we did spectral tests for the alternative combinations of the multipliers. Figure 3 does not suggest any reason to sample any moduli or combinations of them in different proportions, which suggests sampling uniformly. In the process of the investigation though, for various reasons we did run spectral tests for more multipliers with some moduli than others. Rather than throw away some results to make the final search uniform, we present the results of all spectral tests.

Table 2 presents the spectral test results for the 50 best combination generators. In addition to the multipliers and moduli in the underlying generators and the spectral test results, Table 2 also includes the multiplier and modulus of the multiplicative generator which the combination generator approximates. Overall, the results in Table 2 indicate that these combination generators are significantly better than simple multiplicative generators. With a modulus of 2^{31}-1, the multiplier 45991 yields the best spectral test result in eight dimensions among all approximately factorable multipliers. The spectral test result for this multiplier is 0.6984. The best spectral test result for a combined generator in eight dimensions is 0.7616. While 0.76 may not appear much different than 0.70, this appearance is misleading because each spectral test result is the distance relative to the best possible given the modulus. The modulus for the combination generator is effectively on the order of 2^{31} bigger than the modulus for the multiplicative generator and the actual distance is correspondingly smaller. Our best generator also is significantly better than combination generators previously available. The spectral test result for the combination generator in L Ecuyer (1988) is 0.39, in L Ecuyer (1997) is 0.70 and in Knuth (1998, Table 1, line 24 and p. 108) is 0.27. We conclude that this search was worthwhile.

IV. PROPERTIES OF SUBSETS

It is important to test subsets of pseudorandom numbers for apparent deviations from the desired uniform distribution. The spectral test examines the properties of the entire sequence of pseudorandom numbers, but it is quite possible for a generator to produce a complete sequence of pseudorandom numbers

that appears fine but subsets exhibit nonrandom properties. The motivation for testing subsets, therefore, is precisely an interest in the behavior of these subsets. While there are many different tests of the consistency of subsets with a uniform distribution, we focus on a set of tests that we believe is informative.[19] These tests seem broad enough to cover many, although certainly not all, of the possible inconsistencies between the pseudorandom numbers and numbers that are independently and identically uniformly distributed.

Tests for Consistency with Independent Uniform Numbers in One Dimension

The simplest and most direct test for uniformity of the pseudorandom numbers is a Kolmogorov-Smirnov (KS) test whether a set of T pseudorandom numbers is consistent with those generated by a uniform distribution (Stuart and Ord 1987, pp. 1187-90). The KS test is a consistent test against non-uniform alternatives: if the hypothesis of a uniform distribution is false, the probability of the test being inconsistent with the uniform distribution approaches one as T approaches infinity.

Looking at one set of T pseudorandom numbers is somewhat informative, but looking at several sets is substantially more informative. A single KS test on a set of T pseudorandom numbers can appear to be consistent with a uniform distribution even though several tests yield cumulative probabilities, or f-values, that invariably are consistent with the uniform distribution an outcome that is inconsistent with a uniform distribution. A KS test of the consistency of the f-values with a uniform distribution is useful for guarding against the possibility that the pseudorandom numbers are too consistent with a uniform distribution. If the pseudorandom numbers are consistent with a uniform distribution, the probability

[19] The first papers on testing subsets of pseudorandom numbers are those by Kendall and Babington-Smith (1938, 1939) who introduced the serial test discussed below. Knuth (1998, pp. 61-80 and Niederreiter (1992, pp. 166-68) primarily discuss the tests that we use. Fishman and Moore (1982) implement Chi-square tests on grouped pseudorandom numbers in one to four dimensions. Marsaglia (1985) suggests tests that are similar in spirit to those we use. Dwyer and Williams (1996a) discuss the underlying code.

integral transformation indicates that any f-value, whether 0.90, 0.01 or any other value, is itself equally likely.

Proposition: The Probability Integral Transformation (Stuart and Ord 1987, p. 21)

> Suppose that X has the continuous cumulative distribution function $F_X(X)$. Then the cumulative distribution function of the random variable P generated by $P=F_X(X)$ is $F_P(P) = P$, which is a uniform distribution on [0,1].

Because the pseudorandom number generator can be run many times, it is straightforward to generate a set of *N* tests and see whether the f-values from the set of results are consistent with a uniform distribution. In our experience, setting *N* on the order of 100 appears to be more than adequate.

Tests for Consistency with Independent Uniform Numbers in More than One Dimension

We have implemented several tests of the independence and uniformity of the pseudorandom numbers in more than one dimension. One of these tests is a fairly standard one in economics and finance: the runs test. The other tests examine the consistency of *T* pseudorandom numbers in two or more dimensions with independent, uniformly distributed numbers.

Runs Test The runs test is a common test of pseudorandom number generators that indicates whether the generator produces the number of sequences up and down that would be expected in a sequence of independent observations (Gibbons and Chakraborti, Ch. 3). The runs test examines the pseudorandom numbers for too many or too few sequences of increases or decreases in the pseudorandom numbers. There are many possible tests on runs. The most straightforward one is a test whether the actual number of runs up and down in a set of *T* observations is different than the expected number of runs. The resulting test statistic has an asymptotic normal distribution which yields a cumulative probability or f-value. A KS test on N f-values from the runs test provides a good indicator of the overall performance of the generator. This is by no means the only possible runs test and this one may not be particularly discerning.

A runs test that some known generators fail is a runs test based on the relative frequency of different lengths of runs (Knuth 1998, pp. 66-69). Given the number of observations, the expected number of runs of various lengths is simple to compute. The deviation between the actual and expected number of runs can be used in statistics for runs up or for runs down that have an asymptotic Chi-square distribution. The f-values from a set of N Chi-square statistics can be fed into a KS test for the uniformity of the f-values.

Combinations of Sequential Pseudorandom Numbers The next two tests the serial test and the collision test are based on combinations of sequential pseudorandom numbers. The serial test is a test that the pseudorandom numbers are uniformly distributed in two dimensions. The pseudorandom number generator is used to generate sequential pairs of integers that have the same probability if the numbers are independently and uniformly distributed. The consistency of the pairs with a uniform distribution is examined using a Chi-square test on the deviations between actual and expected counts of the various possible pairs. The pairs of integers are generated from the lower or upper bits (or more simply parts) of the pseudorandom numbers. The probability of occurrence of any of the possible pairs of integers *(q,r)* in a collection of pairs with $q, r \in [0, d-1]$ is determined by the size of q and r. The number of possible different values is determined by the number of bits used.[20] If b bits are used in the generated numbers, then there are 2^b possible different values of q and 2^b different possible values of r, which implies that there are 2^{2b} different combinations and the probability of any particular pair is $1/2^{2b}$. An expected value of the number of pairs in a cell equal to five is a common rule of thumb for Chi-square tests of goodness of fit and we follow it. This implies that the number of combinations generated must be $5 \cdot 2^{b2}$ and the total

[20] In our implementation of the test, either the upper or lower bits of the pseudorandom numbers can be used to create pairs. The maximum number of bits that can be extracted from the pseudorandom number is 8 bits, which implies that the largest integral value of each integer in a pair is 255.

number of pseudorandom numbers generated must be twice that.[21] We run the serial test N times and use the resulting f-values in a KS test for a uniform distribution of those f-values.

The collision test is another test for the consistency of the pseudorandom numbers with a uniform distribution in more than one dimension (Knuth 1998, pp. 70-71). The collision test combines consecutive pseudorandom numbers into composite numbers in such a way that repeat values, called collisions, are relatively unlikely. The number of collisions then is compared to the cumulative distribution function, with either too few or too many collisions suggesting that the consecutive numbers are inconsistent with a uniform distribution.

The collision test forms composite numbers represented by 20 bits. The composite numbers are formed by taking the same number of bits from each of the constituent pseudorandom numbers. This implies that there are 2^{20} (1,048,576) unique values of the composite numbers. If it is desired to test for independence and uniformity in two dimensions, ten bits from two sequential pseudorandom numbers are combined to create the 20-bit number. Five bits from four consecutive pseudorandom numbers can be used for a four-dimensional test. Four bits from five consecutive pseudorandom numbers can be used for a five-dimensional test, and two bits from 10 consecutive pseudorandom numbers can be used for a ten-dimensional test.

While repeated composite numbers are not common, their occurrence is not especially rare either. We compute 2^{14} (16,384) composite numbers and then compute the number of repeats of the 2^{20} possible composite numbers. With 2^{14} composite numbers, the number of collisions with a probability of one percent or more is relatively small: between 101 and 154.

---

[21] For example, if the test uses the first 4 bits of each pseudorandom number, the test must use at least 2560 pseudorandom numbers. If the test uses the first 6 bits of each pseudorandom number, then the test must use at least 40,960 pseudorandom numbers and if the test uses the first 8 bits, then the test must use at least 655,360 pseudorandom numbers.

The distribution for the collision test is based on the distribution of the number of empty cells, an occupancy distribution (Johnson, Kotz and Kemp 1992, pp. 414-20) Knuth (1998, p. 71) provides an efficient algorithm to compute the cumulative probabilities of any number of collisions. Because of the discreteness of the number of collisions and therefore the f-values for the number of collisions, we do a Chi-square test on the resulting f-values, with the f-values combined into five classes.

Functions of Pseudorandom Numbers The next two tests the permutation test and the maximum-of-t test are based on functions of the pseudorandom numbers. The permutation test examines the rank order in which sets of pseudorandom numbers are generated, looking for too few or too many repeats of rank orderings, again testing the null hypothesis that sequential numbers appear to be uniformly distributed. In a sequence of t elements, there are $t!$ possible rank orderings with a probability of $1/t!$ for each ordering. A count of the number of occurrences of each ordering of the t elements is used in a Chi-square test of the number of counts relative to the expected number. Because it is desirable to have an expected value of five for each class and there are $t!$ possible classes, the number of pseudorandom numbers required for a test with sets of t elements is at least $5t \cdot t!$.[22] We run the permutation test N times and use the resulting f-values in a KS test for a uniform distribution of the f-values from the N permutation tests.

The maximum-of-t test is another test for uniformity of the pseudorandom numbers in t dimensions but it uses the maximum of the t values instead of the rank order. The maximum value from each set of t numbers is combined with other maximums to create a new series. Knuth (1998, p. 70) derives the probability distribution function if the pseudorandom numbers are independently and uniformly distributed. We do a KS test to compare the distribution of the actual maximum values with the distribution for

[22] For example, if each sequence of pseudorandom numbers has three elements (the smallest number considered), the test requires at least 90 pseudorandom numbers. If each sequence of pseudorandom numbers has 5 elements, the test requires 3000 pseudorandom numbers.

independently and uniformly distributed pseudorandom numbers and then apply a KS test to the resulting f-values.

Normality and Serial Correlation The tests above have examined uniform pseudorandom numbers, although economists commonly use pseudorandom numbers as independent normally distributed numbers. The KS test for consistency with a uniform distribution is equally informative about consistency with a normal distribution. A KS test for consistency with a normal distribution would require transforming the uniform pseudorandom numbers to normally distributed numbers and then using the normal distribution to transform the normally distributed numbers into the associated cumulative probability. These transforms are inverses of each other and the results of a KS test for consistency with a uniform distribution are exactly the same as the results of a KS test for consistency with a normal distribution if the transforms are inverses up to machine precision.[23]

A lack of serial correlation of the normally distributed residuals is a property that is heavily used in economic applications. We transform the uniformly distributed pseudorandom numbers to normally distributed numbers and present Box-Ljung tests of the hypothesis that the first K serial correlations of the normally-distributed numbers are zero. As in our other tests, we use the f-values of the Box-Ljung tests in a KS test for consistency with a uniform distribution.

Test Results Table 3 shows a sample of the results for these tests for the 50 best generators based on the spectral test.[24] Overall, the results are consistent with the hypothesis that the pseudorandom numbers from the generators are consistent with independent, identically distributed numbers and the Box-Ljung serial

---

[23] There are at least two readily available transforms that are machine-precision inverses. One set of these routines is *ndtr* and *ndtri* by Stephen L. Moshier and available at this writing at Netlib (http://www.netlib.org). The other set is *nprob* and *ppnd7* available at StatLib (http://lib.stat.cmu.edu), which we have tested using a version translated from FORTRAN to C by f2c, a program available at Netlib.

[24] Test results for other values of the tests parameters are available from the authors and are consistent with the sample in the table.

correlation test is consistent with serially uncorrelated pseudorandom numbers. As is to be expected in such a large number of tests, some of the p-values of the test statistics are relatively small and some are relatively large. Actually, it would be troubling if none of the generators had p-values in the tails: p-values from a test on a particular generator have a uniform distribution if the null hypothesis is true. This is cold comfort though in the face of a p-value of 4 percent for the maximum-of-t test for the best generator. Does this result indicate that it is better to use another generator besides the first one? Taken at face value, this p-value suggests that the underlying distribution of f-values from the maximum-of-t test is not consistent with a independent uniform distribution. An optimal solution to this problem would use sequential sampling to determine the answer. Instead of pursuing this elaboration, we merely note that the p-value of a KS test on 100 f-values from the maximum-of-t test in Table 3 is 0.433. Overall, we conclude that the first combination generator appears to be quite adequate.

V. ARE ALL GENERATORS ADEQUATE?

Not all readily available generators are adequate. Consider the generators included in the libraries with the Microsoft C^{++} version 4.2 and Borland C^{++} version 4.5 compilers. These generators use a modulus of 2^{32}. The congruential generators themselves produce 32-bit unsigned integers, but the implementations use the upper bits to return a 15-bit signed integer. Table 4 shows the p-values from running the collision test on these generators. For comparison, the table also presents the results for our best combination generator. All of the p-values are from a Chi-square goodness-of-fit statistic on f-values from 100 runs of the collision test. The p-values uniformly indicate that the lower bits are far from random and are quite periodic for generators with a modulus of 2^{32}. The collision test for Microsoft s generator for consecutive pseudorandom numbers based on the lower five bits produces 847 collisions in one test, 887 in the next, 847 in the following one, and so on. This lack of randomness in the lower bits of the numbers produced by congruential generators mod 2^{32} is well known and, indeed, this is the reason that only the

upper 15 bits are produced by these generators.[25] The lower 16 bits are presumed to be sufficiently non-uniformly distributed that it is better to suppress them and use only the upper 15 bits. The p-values in the table indicate, however, that even the lower bits of the upper 15 bits are far from uniformly distributed. This is the case for the best multiplier based on an exhaustive search based on the spectral test (Fishman 1990), not just for the multipliers in these commercial compilers. This best multiplier is as flawed as the ones in the commercial compilers, whether all 31 bits or only the upper 15 bits are used.

The point of this analysis is not to suggest that all congruential generators with a modulus of 2^{32} are worthless or to indicate when they may be adequate. It is beyond the scope of this article to address these issues. These readily available generators do appear to have serious deficiencies though when compared to generators with prime moduli or combination generators.

VI. CONCLUSION

We use four criteria for evaluating pseudorandom number generators: consistency of the computer program s final output with the underlying mathematical model; portability of the computer code; length of the generator s period; and speed. To the extent that a general analysis can evaluate these criteria, the congruential generators analyzed in this paper fare well. For a quick analysis that does not require more than a couple of hundred thousand evaluations of the generator, the multiplicative congruential generator with the best multiplier based on the spectral test is quick and portable when implemented using Hörmann and Derflinger s (1993) algorithm.

For more intensive work, we suggest the combination generator using our best multipliers. It is impossible to verify the consistency of every computer program s final output with the underlying mathematical models. Still, the structure of the phase diagrams of the entire sequence of pseudorandom

---

[25] As Knuth (1998, p. 13) indicates, any set of b lower bits of a generator mod 2^n, $b>n$, behave identically the same as a generator mod 2^b. Hence, the lower 16 bits of a generator mod 2^{32} behave identically to the output of a congruential generator with a modulus of 2^{16}. The cycle length of these lower 16 bits is at most 2^{16}-1.

numbers from a combination generator is known, and that structure can be used to pick combination generators whose full sequences appear to be more uniformly distributed. The spectral test uses that structure to pick the parameter values for the best generators. In applications, these generators produce subsets of these full sequences that are used as if they were draws from some distribution function. We examine the properties of subsets by statistical tests. The tests in this paper include tests for the consistency of the pseudorandom numbers with the underlying distribution, tests for serial correlation of normally-distributed pseudorandom numbers, runs tests and more specialized tests. To the extent possible at a general level, these tests are likely to find generators that will perform poorly in economic research. These tests do not turn up any problems with the best combination generators. An analysis of two generators available with commercial compilers finds a problem with them and indicates that these tests are not vacuous.

The combination generators also fare well on the other criteria. The computer code available with this paper will work in any environment that has 32-bit signed integers. A full cycle from the portable combination generator is about $2.31 \cdot 10^{18}$, which would take about 15,000 years to compute on a Pentium 400. Knuth (1998, p. 185) suggests using no more one-thousandth of a full cycle, but even this limits analysis to numbers that would take 1.5 years to generate. Furthermore, the analysis in the appendix indicates that the suggested algorithm is reasonably quick relative to available alternatives.

We are not suggesting that the combination congruential generator is the best one or always should be used. In fact, congruential generators are poor choices for encryption (Boyar 1989; Krawczyk 1992). When programming, the combination generators in this article are likely to be fine for economic research. There is an enormous temptation to improve a generator by souping it up. However, as Knuth notes (1998, p. 26), One of the common fallacies encountered in connection with random number generation is the idea that we can take a good generator and modify it a little, in order to get an even more random sequence. Such modifications are unwise because [t]his is often false.

We suggest caution concerning pseudorandom number generators. When using a package with a built-in generator, we intend to pursue the following strategy. It is worthwhile to find out what generator is being used in the package, read the description and look for references to published papers about its adequacy. If such a description is not provided with the package s documentation, it is even more important to find out the generator used and references. It may not be possible to acquire this information. In the process of writing this article, we asked for information about the generators in MatLab and Gauss to write routines to test their output and were told that the details were proprietary. Our inclination in such circumstances is to replace the generator. Knowing the generator and being able to program it is necessary to have reproducible results. Besides, the package s creators almost surely have fallen prey to the temptation to invent a generator or modify one; otherwise, the generator would not be proprietary. If the technique is not examined in the open literature, it is hard for non-specialists to be comfortable with the modifications. An alternative is to use tests whose implementation does not use the generator itself, such as Diehard (McCullough 1998).

Whether programming or using a package, it is prudent to test a simulation program including the pseudorandom number generator on a problem with a known answer. If there is reason to be dubious about the generator, the combination generator is relatively simple to program in almost any environment to check the results.

REFERENCES

Boyar, Joan. "Inferring Sequences Produced by Pseudo-Random Number Generators." *Journal of the Association for Computing Machinery* 36 (January 1989): 129-41.

Cassels, J. W. S. *An Introduction to the Geometry of Numbers*. Second printing, corrected. Berlin: Springer-Verlag, 1971.

Dwyer, Jerry [Jr., Gerald P.] "Quick and Portable Random Number Generators." *C/C++ Users Journal* 13 (June 1995), pp. 33-44.

____, and K. B. Williams. (1996a) "Testing Random Number Generators." *C/C++ Users Journal* 14 (June 1996), pp. 39-48.

_____, and _____. (1996b) "Testing Random Number Generators, Part 2." *C/C++ Users Journal* 14 (August 1996), pp. 57-66.

Ferrenberg, Alan M., and D. P. Landau. "Monte Carlo Simulations: Hidden Errors from 'Good' Random Number Generators." *Physical Review Letters* 69 (December 7, 1992), pp. 3382-84.

Fishman, George S. "Multiplicative Congruential Random Number Generators with Modulus $2^{\$}$: An Exhaustive Analysis for $\$=32$ and a Partial Analysis for $\$=48$." *Mathematics of Computation* 54 (January 1990), pp. 331-34.

____. *Monte Carlo*. New York: Springer, 1996.

_____, and Louis R. Moore. "A Statistical Evaluation of Multiplicative Random Number Generators with Modulus 2^{31}-1." *Journal of the American Statistical Association* 77 (March 1982), pp. 129-36.

____, and ____. "An Exhaustive Analysis of Multiplicative Congruential Random Number Generators with Modulus 2^{31}-1." *SIAM Journal on Scientific and Statistical Computing* 7 (January 1986), pp. 24-45.

Gibbons, Jean Dickenson, and Subhabrata Chakraborti. *Nonparametric Statistical Inference*. Third Edition, Revised and Expanded. New York: Marcel Dekker, Inc., 1992.

Giblin, Peter. *Primes and Programming: An Introduction to Number Theory with Computing*. Cambridge: Cambridge University Press, 1993.

Hörmann, W., and G. Derflinger. "A Portable Random Number Generator Well Suited for the Rejection Method." *ACM Transactions on Mathematical Software* 19 (December 1993), pp. 489-95.

Johnson, Norman L., Samuel Kotz and Adrienne W. Kemp. *Univariate Discrete Distributions*. Second edition. New York: John Wiley & Sons, Inc., 1992.

Kendall, M. G., and B. Babington-Smith. "Randomness and Random Sampling Numbers." *Journal of the Royal Statistical Society*, (XXXX 1938), pp. 147-66.

____, and ____. "Second Paper on Random Sampling Numbers." *Supplement to the Journal of the Royal Statistical Society* 6 (No. 1, 1939), pp. 51-61.

Kim, Sangjoom, Neil Shephard and Siddhartha Chib. "Stochastic Volatility: Likelihood Inference and Comparison with ARCH Models." Unpublished paper, Washington University, 1996.

Knuth, Donald E. *The Art of Computer Programming*. Volume 2, Seminumerical Algorithms. Third edition. Reading, Massachusetts: Addison-Wesley Publishing Company, 1998.

Krawczyk, Hugo. "How to Predict Congruential Generators." *Journal of Algorithms* 13 (1992): 527-45.

L Ecuyer, Pierre. "Efficient and Portable Combined Random Number Generators." *Communications of the ACM*. 31 (June 1988), pp. 742-49, 774.

____. "Combined Multiple Recursive Random Number Generators." *Operations Research*, 44 (September-October 1996), pp. 816-22.

____. "Bad Lattice Structures for Vectors of Non-Successive Values Produced by Some Linear Recurrences." *INFORMS Journal on Computing*. 9 (Winter 1997), pp. 57-60.

_____, and Serge Côté. "Implementing a Random Number Package with Splitting Facilities." *ACM Transactions on Mathematical Software* 17 (March 1991), pp. 98-111.

____, and Raymond Couture. "An Implementation of the Lattice and Spectral Tests for Linear Congruential Generators." *INFORMS Journal on Computing*. 9 (Spring 1997), pp. 206-17.

Marsaglia, George. "A Current View of Random Number Generators." In *Computer Science and Statistics, Proceedings of the Sixteenth Symposium*, edited by L. Ballard, pp. 3-10. 1985.

____, and Arif Zaman. "A New Class of Random Number Generators." *Annals of Applied Probability* 1 (3, 1991), pp. 462-80.

McLeod, Ian. "A Remark on Algorithm AS 183. An Efficient and Portable Pseudo-Random Number Generator." *Applied Statistics* 34 (2, 1985): 198-200.

McCullough, Bruce. "Econometric Software Reliability: EViews, LIMDEP, SHAZAM and TSP." *Journal of Applied Econometrics*, 1999, forthcoming.

Monahan, John F. "Accuracy in Random Number Generation." *Mathematics of Computation* 45 (October 1985): 559-68.

Niederreiter, Harald. *Random Number Generation and Quasi-Monte Carlo Methods*. CBMS-NSF Regional Conference Papers in Applied Mathematics, volume 63. Philadelphia: Society for Industrial and Applied Mathematics, 1992.

Park, Stephen K., and Keith W. Miller. "Random Number Generators: Good Ones Are Hard to Find." *Communications of the ACM* 31 (October 1988), pp. 1192-1201.

Payne, Jr., W. H., J. R. Rabung and T. P. Bogyo. "Coding the Lehmer Pseudo-random Number Generator." *Communications of the ACM* 12 (February 1969), pp. 85-86.

Press, William H., Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery. *Numerical Recipes in C*.

Second edition. Cambridge: Cambridge University Press, 1992.

Schmid, F., and N. B. Wilding. "Errors in Monte Carlo Simulations Using Shift Register Random Number Generators." *International Journal of Modern Physics C* 6 (6, 1995), pp. 781-87.

Schrage, Linus. "A More Portable Fortran Random Number Generator." *ACM Transactions on Mathematical Software* 5 (June 1979), pp. 132-38.

Stuart, Alan, and J. Keith Ord. *Kendall's Advanced Theory of Statistics*. New York: Oxford University Press, 1991.

Wichman, B. A., and I. D. Hill. "An Efficient and Portable Pseudo-random Number Generator." *Applied Statistics* 31 (2, 1982): 188-90.

_____, and _____. "An Efficient and Portable Pseudo-random Number Generator: Correction." *Applied Statistics* 33 (1, 1984): 123.

Zeisel, H. "A Remark on Algorithm AS 183. An Efficient and Portable Pseudo-Random Number Generator." *Applied Statistics* 35 (1, 1986): 89.

Figures 1–3

Click on the links below to open Figures 1–3. Be aware that these files are quite large; the file size is indicated in parentheses.

- Figure 1 (1,348 KB)
- Figure 2 (1,315 KB)
- Figure 3 (1,169 KB)

Table 1

SPECTRAL TESTS AND RESULTS

Modulus		Modulus		Number of Trials	Fraction of Trials
2^{31}-1	2147483647	2^{31}-19	2147483629	26048975	0.164
		2^{31}-61	2147483587	16004883	0.101
		2^{31}-69	2147483579	4388455	0.028
		2^{31}-85	2147483563	4140000	0.026
		2^{31}-99	2147483549	5039152	0.032
		2^{31}-105	2147483543	39075500	0.246
2^{31}-19	2147483629	2^{31}-61	2147483587	3000000	0.019
		2^{31}-69	2147483579	3000000	0.019
		2^{31}-85	2147483563	3000000	0.019
		2^{31}-99	2147483549	3000000	0.019
		2^{31}-105	2147483543	6000000	0.038
2^{31}-61	2147483587	2^{31}-69	2147483579	13172284	0.083
		2^{31}-85	2147483563	4473228	0.028
		2^{31}-99	2147483549	3000000	0.019
		2^{31}-105	2147483543	3000000	0.019
2^{31}-69	2147483587	2^{31}-85	2147483563	3000000	0.019
		2^{31}-99	2147483549	3000000	0.019
		2^{31}-105	2147483543	3000000	0.019
2^{31}-85	2147483563	2^{31}-99	2147483549	6000000	0.038
		2^{31}-105	2147483543	3000000	0.019
2^{31}-99	2147483549	2^{31}-105	2147483543	4222502	0.027

Table 2

BEST COMBINATION MULTIPLIERS

First		Second		Lowest Spectral Test Results	Dimension	Combination Multiplier	Combination Modulus
Multiplier	Modulus	Multiplier	Modulus				
65670	2147483647	44095	2147483587	0.7616092	8	384306384907687752	4611685885283401789
28078	2147483543	2568	2147483629	0.7587240	6	2359467766005139171	4611685752139417547
67142	2147483579	78375	2147483563	0.7548043	7	3746996128936123305	4611685687714911977
75756	2147483647	104165	2147483629	0.7536803	5	3330665482726365875	4611685975477714963
19391	2147483647	15514	2147483629	0.7513183	8	1793432972363903020	4611685975477714963
17916	2147483587	342720	2147483549	0.7509227	6	2184501043636838355	4611685674830010263
19995	2147483647	172074	2147483543	0.7507221	8	3148365555130290821	4611685790794121321
7332	2147483647	5557	2147483587	0.7503238	7	384307093577232924	4611685885283401789
164130	2147483587	44888	2147483579	0.7500295	7	3458732295661317403	4611685739254517873
56599	2147483647	75939	2147483543	0.7491809	4	3015333416407114126	4611685790794121321
67422	2147483587	550072	2147483563	0.7490791	7	384350328864351860	4611685704894780481
73865	2147483587	39810	2147483579	0.7490010	6	2882294445464953125	4611685739254517873
22313	2147483563	54764	2147483549	0.7487799	5	329411093662844313	4611685623290405087
502805	2147483587	48019	2147483543	0.7485635	5	3982797238828764959	4611685661945108741
14088	2147483647	13632	2147483543	0.7478362	6	2483215416396343424	4611685790794121321
77694	2147483647	54218	2147483629	0.7473586	8	1024816304865897854	4611685975477714963
63358	2147483647	9609	2147483549	0.7473015	2	799985135076031234	4611685803679023203
489845	2147483647	24682	2147483543	0.7470770	8	2350176422905804658	4611685790794121321
91608	2147483587	15913	2147483579	0.7469910	5	2882283267812900533	4611685739254517873
42716	2147483587	29628	2147483579	0.7466388	4	4611682225971412257	4611685739254517873
58333	2147483579	89251	2147483549	0.7465280	8	2767013607786719973	4611685657650141871
270907	2147483587	63627	2147483579	0.7464680	4	4611630097955049610	4611685739254517873
40016	2147483647	51804	2147483587	0.7461276	3	4304240581506835533	4611685885283401789
217180	2147483587	11849	2147483579	0.7458094	7	576405599287981627	4611685739254517873
24929	2147483629	11778	2147483587	0.7455880	8	1866634075026945115	4611685846628697223
88794	2147483629	82062	2147483579	0.7455030	7	737869443574705496	4611685829448828191
748	2147483629	14433	2147483549	0.7453926	8	3170534330808138567	4611685765024319321
27725	2147483629	26746	2147483587	0.7448360	4	2086238785322855688	4611685846628697223
88464	2147483647	37822	2147483629	0.7447427	6	2049632169497581149	4611685975477714963
113864	2147483629	36301	2147483549	0.7444434	7	403520422373832853	4611685765024319321
135291	2147483647	38747	2147483543	0.7442158	7	1064233188965497750	4611685790794121321
39044	2147483647	48161	2147483629	0.7441394	8	2305844075439363731	4611685975477714963
51827	2147483587	43765	2147483563	0.7440304	5	1921534988997348761	4611685704894780481
58334	2147483579	40345	2147483563	0.7438494	5	4323452917790156898	4611685687714911977
36169	2147483647	109733	2147483563	0.7438046	4	1098022317290070611	4611685833743794261
48060	2147483629	39069	2147483587	0.7437986	8	2964654727403607981	4611685846628697223
55304	2147483647	47776	2147483543	0.7437353	5	2483215270367496644	4611685790794121321
438799	2147483647	163368	2147483543	0.7437206	3	2882297931904761001	4611685790794121321
31915	2147483647	359531	2147483543	0.7436792	6	1773732069055487188	4611685790794121321
273181	2147483647	55107	2147483629	0.7436263	6	1024793088420385624	4611685975477714963
17722	2147483647	119278	2147483543	0.7435447	8	2305844992414859678	4611685790794121321
490629	2147483543	934094	2147483629	0.7432271	5	1018849732056996872	4611685752139417547
892182	2147483647	77203	2147483629	0.7431267	7	2818155309788654192	4611685975477714963
58235	2147483647	10829	2147483543	0.7423128	5	2571900712138993494	4611685790794121321
81270	2147483647	27650	2147483549	0.7422231	8	658811082688174869	4611685803679023203
48986	2147483647	693	2147483543	0.7419447	6	3680479008918167304	4611685790794121321
2207074	2147483647	21606	2147483587	0.7419340	7	307367504741249830	4611685885283401789
129063	2147483647	171005	2147483543	0.7418388	7	2749275087490462182	4611685790794121321
31481	2147483579	40419	2147483563	0.7417646	3	576461910602409797	4611685687714911977
66551	2147483543	22408	2147483629	0.7415265	5	3056583519399053049	4611685752139417547

Table 3
A SAMPLE OF THE RESULTS OF STATISTICAL SUBSEQUENCE TESTS
p-values

Frequency	Runs	Serial[a]		Collision[b]		Permutation[c]	Maximum of T[d]	Serial Correlation[e]
		Upper	Lower	Upper	Lower			
0.2031	0.8111	0.9278	0.0849	0.3926	0.7358	0.5415	0.0368	0.2269
0.0291	0.8911	0.4379	0.8500	0.6268	0.4628	0.6372	0.6678	0.4117
0.5921	0.2887	0.9359	0.7040	0.6268	0.3669	0.5559	0.5990	0.4689
0.8768	0.8188	0.4631	0.7830	0.4779	0.2674	0.8012	0.8899	0.3993
0.4236	0.7852	0.4216	0.6013	0.2227	0.5747	0.1906	0.7958	0.2634
0.6247	0.2150	0.5872	0.0264	0.4628	0.7725	0.2262	0.3995	0.3348
0.2034	0.2182	0.5111	0.4374	0.4481	0.0404	0.2657	0.3159	0.2338
0.0830	0.2022	0.3581	0.8715	0.5089	0.2067	0.9441	0.8552	0.5215
0.9335	0.2154	0.2329	0.6616	0.4197	0.7358	0.5286	0.0535	0.4802
0.9015	0.3119	0.2007	0.7250	0.0113	0.6446	0.4853	0.9013	0.6822
0.4737	0.7780	0.8259	0.9011	0.1712	0.0357	0.7329	0.2751	0.1432
0.6099	0.8756	0.5281	0.1979	0.5249	0.5089	0.3749	0.7566	0.1070
0.2561	0.5646	0.1423	0.0697	0.4337	0.4337	0.1722	0.6218	0.3047
0.3056	0.2843	0.3701	0.3801	0.1778	0.0477	0.4200	0.6382	0.0977
0.0607	0.0139	0.9222	0.8143	0.3084	0.0022	0.0427	0.6333	0.1966
0.5039	0.1217	0.8363	0.6146	0.7174	0.7541	0.5571	0.7662	0.2962
0.4919	0.0497	0.4936	0.4274	0.6808	0.3796	0.7169	0.4124	0.8241
0.8702	0.1551	0.9953	0.6885	0.5918	0.9513	0.9142	0.4248	0.5301
0.3559	0.5324	0.0681	0.9484	0.5412	0.5412	0.4658	0.8064	0.0460
0.6955	0.5538	0.7770	0.2353	0.1918	0.8266	0.7392	0.7709	0.7943
0.9747	0.4123	0.3994	0.4995	0.5918	0.0255	0.4721	0.6732	0.8953
0.3075	0.2113	0.9076	0.6525	0.0636	0.6092	0.8418	0.6100	0.5067
0.0282	0.0348	0.1185	0.1761	0.1778	0.6808	0.8033	0.4299	0.2259
0.7642	0.8501	0.7314	0.4664	0.2487	0.3796	0.5519	0.0468	0.5692
0.8089	0.5812	0.6608	0.9517	0.9246	0.3546	0.8035	0.3620	0.6041
0.3186	0.6611	0.6554	0.1209	0.4481	0.3669	0.5462	0.2858	0.4749
0.3190	0.5320	0.3434	0.9247	0.5747	0.2674	0.1178	0.7521	0.7653
0.8390	0.5195	0.9156	0.8745	0.8266	0.5747	0.3678	0.4819	0.6356
0.9962	0.3276	0.3309	0.0784	0.3926	0.3084	0.9880	0.7963	0.6336
0.8204	0.1218	0.2560	0.5795	0.2772	0.1847	0.9259	0.4189	0.9872
0.2968	0.8858	0.8672	0.7965	0.6990	0.2772	0.1160	0.1509	0.5616
0.9035	0.2476	0.9328	0.9943	0.5412	0.0388	0.0201	0.2080	0.0716
0.4208	0.0072	0.1382	0.4216	0.8614	0.2873	0.4139	0.9618	0.3806
0.3761	0.4799	0.4672	0.4383	0.0477	0.9898	0.0192	0.7843	0.0529
0.6576	0.0385	0.6485	0.1870	0.9735	0.5412	0.5489	0.7755	0.3165
0.1424	0.8606	0.6549	0.7158	0.1209	0.2674	0.3405	0.4499	0.5777
0.8244	0.8981	0.4977	0.8857	0.0215	0.1359	0.4183	0.9578	0.9314
0.4938	0.9407	0.7040	0.2467	0.0812	0.3546	0.9238	0.3729	0.0536
0.4160	0.0715	0.4058	0.6449	0.2674	0.0342	0.6860	0.0956	0.0689
0.9855	0.7530	0.6763	0.8684	0.3084	0.6808	0.3459	0.8966	0.1644
0.0378	0.0725	0.4107	0.3819	0.4197	0.4481	0.7039	0.2211	0.5895
0.9466	0.4613	0.7322	0.7686	0.9098	0.8266	0.5309	0.6254	0.1366
0.9117	0.1355	0.0813	0.9725	0.9513	0.7907	0.1461	0.5304	0.7203
0.0913	0.4866	0.3943	0.2185	0.4779	0.2873	0.7337	0.0859	0.7104
0.4034	0.3667	0.2614	0.3567	0.3309	0.5918	0.4074	0.4891	0.1130
0.0465	0.5372	0.1812	0.0550	0.4481	0.9098	0.1292	0.6278	0.5197
0.1124	0.4757	0.3180	0.7576	0.6446	0.3195	0.9594	0.2635	0.3081
0.7678	0.0068	0.4092	0.8915	0.3084	0.7907	0.9620	0.4185	0.6566
0.6786	0.0101	0.5650	0.6898	0.0636	0.8614	0.2026	0.3869	0.5470
0.2008	0.6270	0.8223	0.4424	0.8943	0.7725	0.0063	0.3898	0.9277

a. The serial test results are for 5 bits from each number.
b. The collision test results are for combinations of 5 numbers into a single 20-bit value.
c. The permutation test results are for permutations of 5 numbers.
d. The maximum-of-t test results are for the maximum of 5 numbers.
e. The serial correlation test results are for 10 serial correlations.

Table 4
COLLISION TESTS ON GENERATORS MOD 2^{32}
100 RUNS OF COLLISION TEST
p-values

NUMBERS COMBINED/BITS	MICROSOFT C++ VERSION 4.2 15 BITS	BORLAND C++ VERSION 4.5 15 BITS	FISHMAN (1996) 31 BITS	FISHMAN (1996) 16 BITS	BEST COMBINATION 31 BITS
2/upper 10	0.231	0.448	0.141	0.627	0.092
2/lower 10	0.974	0.558	0.000	0.012	0.258
4/upper 5	0.215	0.844	0.056	0.681	0.736
4/lower 5	0.000	0.000	0.000	0.000	0.502
5/upper 4	0.558	0.240	0.451	0.240	0.393
5/lower 4	0.000	0.000	0.000	0.000	0.756
10/upper 2	0.343	0.131	0.681	0.592	0.525
10/lower 2	0.000	0.000	0.000	0.000	0.861

Appendix
Implementing the Generators

Computation of pseudorandom numbers with congruential generators can overflow. For example, suppose that the largest signed integer represented by a native type in a language is 2^{31}-1, a common situation on 32-bit computers. Also suppose that the modulus for a multiplicative generator is 2^{31}-1 and that the multiplier is chosen so that the generator has the maximum period of [1, 2^{31}-2], for example the multiplier 45991. All these integers are representable in the native types in the language. At some point, the value of x_{i-1} is 2^{31}-2. The value of the next pseudorandom number is computed by first multiplying 2^{31}-2 by 45991, which yields a value greater than 2^{31}-1, the maximum representable positive value.

There are solutions to this problem. One possibly obvious solution is to switch to floating-point arithmetic. Without care though, loss of significant intermediate digits and eventual errors in the computations are quite possible.

Approximate factoring is a relatively quick way to compute the results from congruential generators without loss of precision. The multiplicative generator is

$$x_i = ax_{i-1} \bmod m$$

with a and x positive and m greater than either a or any value of x. If integer overflow were not an issue, the residue x_i could be computed from $x_i = ax - k_1 m$, where $k_1 = \lfloor ax_{i-1} / m \rfloor$ which is the largest integer less than or equal to ax_{i-1} / m.

Suppose that the intermediate value ax would overflow. The modulus m always can be approximately factored into $m = aq + r,$ where $q = \lfloor m / a \rfloor$. Now calculate $k = \lfloor x_{i-1} / q \rfloor$, which avoids overflow associated with computing k_1 from by $k_1 = \lfloor ax_{i-1} / m \rfloor$. Because $m = aq + r,$

$$\begin{aligned} x_i &= ax_{i-1} - k_1 m \\ &= ax_{i-1} - km + (k - k_1)m \\ &= a(x_{i-1} - kq) - kr + (k - k_1)m. \end{aligned}$$

Approximate factoring is not the only way to avoid overflow. Some algorithms, based on an

algorithm by Payne, Rabung and Bogyo (1969), exploit simplifications possible with a modulus of 2^b-1 in computers with a word size of $b+1$ bits or more. Given 32-bit signed integers, such algorithms work for a modulus of 2^{31}-1. Fishman (1996, pp. 602-607) provides an implementation in FORTRAN using floating point computations with 47 bits of precision. Hörmann and Derflinger (1993) provide implementations in ANSI C and FORTRAN that work for multipliers less than 2^{30}.

Another algorithm is quite general and imposes no restrictions on multipliers. L Ecuyer and Côté (1991, p. 104) decompose the 32-bit multiplier and modulus into upper and lower halves and perform computations on these 16-bit quantities to generate the 32-bit result for a congruential generator.

Table A1 provides some evidence on the speed of these various routines on computers and operating systems available to us. Decomposition works for general moduli and multipliers although it can take ten times longer than another algorithm. Approximate factoring is faster than decomposition for those multipliers that can be approximately factored. For a multiplicative generator with a modulus of 2^{31}-1, Hörmann and Derflinger s routine is significantly faster than approximate factoring besides being less restrictive concerning the multipliers that can be used. In the table, Fishman s floating-point routine is slower than most of the other routines. The speed differences depend partly on compiler optimization. With no compiler optimization for any algorithm, Fishman s routine is fastest.

What is the best way to generate pseudorandom numbers? Multiplicative congruential generators can be sufficient for some purposes. We recommend using Hörmann and Derflinger s FORTRAN routine or a translation to another language with a multiplier of 742938285 and a modulus of 2^{31}-1 (Fishman and Moore 1986, p. 43). Such a routine is generally fast. This preference is reinforced by Hörmann and Derflinger s evidence (1993) that pseudorandom numbers computed using approximate factoring do not produce good approximations to non-uniform distributions when transformed by the rejection method. We provide one version in C^{++} in Listing 1. Listing 2 uses operators available in C and C^{++} but not FORTRAN or other languages such as BASIC or Pascal. Hörmann and Derflinger s method is not

available for a modulus other than 2^{31}-1 and there are circumstances in which such a modulus might be useful. If the rejection method will be used to generate non-uniformly distributed pseudorandom numbers with a modulus other than 2^{31}-1, approximate factoring should not be used and decomposition appears to be the method of choice. If the rejection method will not be used, the choice between approximate factoring and decomposition is a tradeoff of speed for somewhat better spectral test results.

Combination generators have substantial advantages over multiplicative congruential generators. Combination generators have a long enough period that using much of the period cannot be an issue. The spectral test results are better than with a single multiplier. Combination generators can be efficiently implemented using approximate factoring and, as the code in this appendix illustrates, the programming is not particularly complicated. Also, it is easy to use the same generator on different machines and guarantee non-overlapping sampling intervals using the *jump_ahead* routine in Dwyer (1995). The analysis in Hörmann and Derflinger suggests that the problem with the rejection method is unlikely to extend to combination generators with individual generators computed by approximate factoring. Hence, for combination generators, we suggest using approximate factoring for the individual generators. Hörmann and Derflinger s (1993) routine is not as useful for combination generators because it cannot be used for both of the multiplicative generators and we prefer to avoid having multiple algorithms in one program to do similar computations. Decomposition is slower than approximate factoring and this speed difference is more important for the combined generator than for the multiplicative generator. The combined generator requires computation of two pseudorandom numbers at each step and takes slightly more than twice as long as the algorithm to produce the intermediate pseudorandom numbers. A routine in C^{++} to compute a combination generator is provided in Listing 3.

APPENDIX TABLE 1
TIME TO GENERATE 1 MILLION PSEUDORANDOM NUMBERS
Seconds

COMPUTER	L'ECUYER AND CÔTÉ (1991) DECOMPOSITION	APPROXIMATE FACTORING	FISHMAN (1996, P. 604) PAYNE *ET AL.*	HÖRMANN AND DERFLINGER (1993) PAYNE *ET AL.*
Pentium 100 Windows 3.1 MS-DOS box	10.33	2.25	3.35	1.32
Pentium 133 notebook Windows 95	2.20	1.37	2.69	0.22
Pentium 200 Pro Windows NT	1.05	0.38	1.91	0.14
Sun Sun OS 5.4	1.33	0.88	1.48	0.64

The comparisons are useful for comparing orders of magnitude. Seemingly minor variations in operating system, compiler and instruction set on a CPU can be quite important. The MS-DOS version of the program is compiled with Microsoft C/C^{++} version 7.0; the Windows 95 and NT versions are compiled with Microsoft Visual C^{++} version 4.2; the Sun version is compiled with the Sun compiler SC3.0.1. With two exceptions on the Sun, all times are generated by programs compiled with aggressive compiler optimizations. On the Sun, approximate factoring is fastest with some but not all optimizations and Fishman s routine is consisently faster with no optimization than with any optimization. The final pseudorandom number was printed to ensure that the routines were not optimized out of the timing programs. There are two versions of the decomposition algorithm. One version of the decomposition algorithm is a transliteration of L Ecuyer and Côté s implementation in Pascal and the other uses some obvious speedups feasible in C and C^{++} that are not available in Pascal. On the Sun computer, the transliteration of the Pascal routine is faster and on the Intel platform, the speedups improve the timing. We present the faster results for each machine. Hörmann and Derflinger provide two versions of their routines: one in C, and another in FORTRAN. The C routine uses code that relies on the availability of unsigned integers and the FORTRAN routine uses signed 32-bit integers. In the table, we provide times for a C transliteration of the FORTRAN routine, but the times for the C code are little different.

Listing 1

Code Listing for Multiplicative Generator

```
/* PayneH[ormann]D[erflinger].cpp
        RandNum can be initialized outside the file before the first call
      and holds the prior pseudorandom number for each call
   This algorithm works for a modulus of 2147483647L only */

static const long twoT16 = 65536 ;              // 2^16
static const long twoT15 = 32768 ;              // 2^15
static const long mult    = 742938285L ;
static const long multHI  = (mult / twoT15) ;
static const long multLO  = (mult % twoT15) ;         // mult mod 2^15
static const long modulus = 2147483647L ;

long RandNum = 1 ;

long PayneHD(void)
{

        long mid, randhi, randlo, temp ;

        randhi = RandNum / twoT16 ;
        randlo = RandNum % twoT16 ;                             // randlo = rand mod 2^16
        mid = (-modulus + multHI * randlo) + (multLO * 2) * randhi ;
        if (mid < 0)
                mid += modulus ;

        RandNum = (-modulus + multHI * randhi) + (mid / twoT16) + multLO * randlo ;

        if (RandNum > 0)
                RandNum -= modulus ;

        temp = mid % twoT16 ;                                   // temp = mid mod 2^16
        RandNum += temp * 32768 ;

        if (RandNum < 0)
                RandNum += modulus ;

        return RandNum;
}
```

Listing 2

Code Listing for Multiplicative Generator Using Operators Available in C and C++

```
/* PayneH[ormann]D[erflinger].cpp
     RandNum can be initialized outside the file before the first call
          and holds the prior pseudorandom number for each call
    This algorithm works for a modulus of 2147483647L only */


static const long    mult      = 742938285L ;
static const long    multHI    = (mult >> 15) ;
static const long    multLO    = (mult & 0x7fff) ;
static const long    modulus = 2147483647L ;

long RandNum = 1 ;

long PayneHD2(void)
{

     long mid, randhi, randlo ;

     randhi = RandNum >> 16 ;
     randlo = RandNum & 0xffff ;
     mid = (-modulus + multHI * randlo) + (multLO << 1) * randhi ;
     if (mid < 0)
          mid += modulus ;

     RandNum = (-modulus + multHI * randhi) + (mid >> 16) + multLO * randlo ;
     if (RandNum > 0)
          RandNum -= modulus ;
     RandNum += ((mid & 0xffff) << 15) ;
     if (RandNum < 0)
          RandNum += modulus ;

     return RandNum;
}
```

Listing 3
Code for Combination Generator

```
/*      RandComb.cpp
        combination multiplicative random number generator
                takes difference between two random numbers
                rand1 and rand2 can be initialized outside of file */

static const long mod1  = 2147483647 ;
static const long mult1 = 65670 ;
static const long q1    = mod1 / mult1 ;
static const long r1    = mod1 - mult1 * q1 ;

static const long mod2  = 2147483587 ;
static const long mult2 = 44095 ;
static const long q2    = mod2 / mult2 ;
static const long r2    = mod2 - mult2 * q1 ;

long rand1=1, rand2=1 ;

        /* generate the next value in sequence from generator using approximate factoring */
long inline GenrRand(long rand, long modulus, long mult, long q, long r)
{
        long temp ;

        temp = rand / q ;
        temp = mult * (rand - temp * q) - temp * r ;
        if (temp < 0)
                temp += modulus ;
        return temp ;
}
        /* get a random number */
long RandComb(void)
{
        long RandNum ;

        rand1 = GenrRand(rand1, mod1, mult1, q1, r1) ;
        rand2 = GenrRand(rand2, mod2, mult2, q2, r2) ;

        RandNum = rand1 - rand2 ;
        if (RandNum < 0)
                RandNum += mod1-1 ;

        return RandNum ;
}
```