RANDOM NUMBERS FOR SIMULATION

RANDOM NUMBERS FOR SIMULATION

PIERRE L'ECUYER

In the mind of the average computer user, the problem of generating uniform variates by computer has been solved long ago. After all, every computer system offers one or more function(s) to do so. Many software products, like compilers, spreadsheets, statistical or numerical packages, etc. also offer their own. These functions supposedly return numbers that could be used, for all practical purposes, as if they were the values taken by independent random variables, with a uniform distribution between 0 and 1. Many people use them with faith and feel happy with the results. So, why bother?

Other (less naive) people do *not* feel happy with the results and with good reasons. Despite renewed crusades, blatantly bad generators still abound, especially on microcomputers [55, 69, 85, 90, 100]. Other generators widely used on medium-sized computers are perhaps not so spectacularly bad, but still fail some theoretical and/or empirical statistical tests, and/or generate easily detectable regular patterns [56, 65].

Fortunately, many applications appear quite robust to these defects. But with the rapid increase in desktop computing power, increasingly sophisticated simulation studies are being performed that require more and more "random" numbers and whose results are more sensitive to the quality of the underlying generator [28, 40, 65, 90]. Sometimes, using a not-so-good generator can give totally misleading results. Perhaps this happens rarely, but can be disastrous in some cases. For that reason, researchers are still actively investigating ways of building generators. The main goal is to design more robust generators without having to pay too much in terms of portability, flexibility, and efficiency. In the following sections, we give a quick overview of the ongoing research. We focus mainly on efficient and recently proposed techniques for generating *uniform pseudorandom numbers*. Stochastic simulations typically transform such numbers to generate variates according to more complex distributions [13, 25]. Here, "uniform pseudorandom" means that the numbers behave from the outside as if they were the values of i.i.d. random variables, uniformly distributed over some finite set of symbols. This set of symbols is often a set of integers of the form $\{0, \ldots, m-1\}$ and the symbols are usually transformed by some function into values between 0 and 1, to approximate the $U(0,1)$ distribution. Other tutorial-like references on uniform variate generation include [13, 23, 52, 54, 65, 84, 89].

Views of Randomness

Classical Definitions

In the classical (Kolmogorov) sense, a string of bits is *random* if it cannot be described by a shorter string than itself. A generalization is that it cannot be produced efficiently (e.g., in polynomial time), by a program smaller than itself. For references and other definitions, see [43, 52]. These definitions do not tell us how to generate such bits on computers.

In the early days, *physical devices* (like noise diodes, Geiger counters, etc.) have been attached to computers with the aim of producing such "true" random bits (see the references in [18]). These methods were abandoned for many reasons, including the following: using such specialized hardware is not convenient; a sequence of numbers cannot be repeated without storing it; and the numbers produced are not necessarily uniformly distributed [12, 18]. Work is still being done on ways to extract "random-looking" bits from imperfect physical sources of randomness [18], but at the present time, these techniques are not practical enough for standard simulation applications.

A Framework for PRNGs

The so-called "random number generators" that are used in practice are in fact deterministic functions that produce a periodic sequence of numbers. When their initial state (called the seed) is truly random, they can be viewed as *extensors* of randomness, whose purpose is to save "coin tosses." They stretch a short truly random seed into a long sequence of values that is supposed to appear and behave like a true random sequence. For this reason, they are often called *pseudorandom*. We now sketch a framework for studying such generators. In [60] a *pseudorandom number generator* (PRNG) is defined as a family $\{G_n, n \geq 1\}$ of structures, increasing in size. This will be discussed a little further in the next subsection, but for the remainder of the paper, we adopt a simplified definition, in which we fix the size (as is always the case in practice). We simply use the term *generator*.

DEFINITION 1. A *generator* is a structure $G = (S,\mu,f,U,g)$, where S is a finite set of *states*, μ is a probability distribution on S, called the *initial distribution*, $f : S \to S$ is the *transition function*, U is a finite set of *output* symbols, and $g : S \to U$ is the *output function*.

A generator operates as follows:

(1) Select the initial state $s_0 \in S$ according to μ; let $u_0 := g(s_0)$;

(2) for $i := 1,2, \ldots$, let $s_i := f(s_{i-1})$ and $u_i := g(s_i)$.

The sequence of *observations* $(u_0, u_1, u_2, \ldots)$ is the output of the generator. The initial state s_0 is called the *seed*. We assume that efficient procedures are available to compute f and g and to generate the seed s_0 according to μ. Sometimes, in practice, the seed is a fixed constant, which means that real randomness is completely eliminated. In other cases, some people would determine the seed by reading for example the computer's clock; this is not necessarily a good idea because it makes replication and debugging hard. Of course, the aim of a generator will be to output a much longer sequence than its input seed s_0. The output sequence should also *look* to some extent (when the seed is random) as if the u_i's were the values of i.i.d. random

variables, uniformly distributed over U. In practice, this should be supported by a sound theoretical basis and assessed empirically by powerful statistical tests. Since S is finite, the sequence of states is ultimately periodic. The *period* is the smallest positive integer ρ such that for some integer $\pi \geq 0$ and for all $n \geq \pi$, $s_{\rho+n} = s_n$. The smallest π that satisfies this is called the *transient*. When $\pi = 0$, the sequence is said to be *purely periodic*.

PT-perfect generators

In an *idealized* generator, nobody using reasonable computing resources and reasonable time, could distinguish between the generator's output and a sequence of truly i.i.d. uniform variates over U better than by flipping a fair coin to guess which is which. Note that this is reminiscent of Turing's test for intelligence.

L'Ecuyer and Proulx [60] (and other references given there) give a more precise definition, based on computational complexity. This definition applies to a family $\{G_n, n \geq 1\}$ of generators. Informally, the family is called *PT-perfect if* G_n "runs" in polynomial-time (in n) and if no polynomial-time (in n) statistical test can distinguish the output of the generator from a truly random sequence (or equivalently, no polynomial-time algorithm can predict u_{i+1} from $(u_0, \ldots, u_i)$) significantly better than by picking a value uniformly from U. Also see [6, 9, 43, 87].

The generators used most often in simulation—linear congruential, multiple recursive, GFSR, . . .—are *not* PT-perfect. "Efficient" algorithms have been designed to infer their sequence by looking at the first few numbers [10, 91]. But in practice, they remain the most useful generators for simulation. They are efficient and show good statistical behavior with respect to most reasonable empirical tests. Binary (or m-ary) expansions of algebraic numbers (roots of polynomials with integral coefficients) or of some transcendental numbers (including π) do not define either PT-perfect generators. Kannan et al. [51] give efficient algorithms to compute further digits given a long enough initial segment of the expansion.

PT-perfect generators were introduced by researchers in cryptology. These people proposed various generators that are *conjectured* to be PT-perfect. Typically, those generators (at least those currently proposed) are much too slow for simulation use. Also, the very existence of any PT-perfect generator has not been proven.

Matrix Linear Congruential Recurrences

Most generators used in practice are based on linear recursions with modular arithmetic. Typically, they are special cases or variants of the following matrix formulation. For some positive integers m and k, let F be the set of integers $\{0,1, \ldots, m-1\}$ and S the set of k-dimensional vectors with components in F, i.e., $S = \{X = (x_1, \ldots, x_k)' \mid x_i \text{ integer and } 0 \leq x_i < m \text{ for } 1 \leq i \leq k\}$. Let $A = (a_{ij})$ be a $k \times k$ matrix with elements in F and $C \in S$ a constant vector. Define a linear transformation $f: S \to S$ by $f(X) = (AX + C) \bmod m$ (where the modulo operation is taken elementwise). Let μ be an initial distribution on S. Choosing μ, U and $g: S \to U$ defines a generator in the sense of Definition 1 (we will examine ways of defining U and g later). Here, the generator's state evolves as

$$X_n := (AX_{n-1} + C) \bmod m, \quad (1)$$

where the initial state X_0 is the *seed*. Equation (1) defines a *linear congruential generator* (LCG) in matrix form. In the simulation literature, the term LCG usually refers to the case $k = 1$, but here, we adopt the more general definition. When $C = 0$ (the most popular case), the generator is called *multiplicative* (MLCG) and (1) becomes

$$X_n := AX_{n-1} \bmod m. \quad (2)$$

Here, obviously, the vector $X_n = 0$ must be avoided. MLCGs in matrix form have been studied, for example, in [3, 44, 45, 58, 76, 83]. In fact, any LCG of order k can be expressed as a MLCG of order $k + 1$ as follows: add to A a $(k + 1)$-th column that contains C, then a $(k + 1)$-th line that contains a 1 in position $(k + 1)$ and zeros elsewhere; add a 1 as the $(k + 1)$-th component of X_n.

Prime Modulus

Suppose m is prime and $C = 0$. (When m is prime, taking $C \neq 0$ has no significant interest.) In that case, F and S can be identified respectively with $GF(m)$ and $GF(m^k)$, where for any $e > 0$, $GF(m^e)$ denotes the Galois field with m^e elements [63, 76]. $GF(m^k)$ can also be viewed as a field of polynomials of degree smaller than k, with coefficients in $GF(m)$. Let S^* be the set obtained by removing the vector 0 from S. The maximal possible period for the X_n's is the cardinality of S^*, i.e., $p = m^k - 1$. It is attained if and only if all powers of A in arithmetic mod m, plus the matrix 0, form a vector space with m^k elements, isomorphic to $GF(m^k)$. An equivalent condition is that the characteristic polynomial of A,

$$f(x) = |xI - A| \bmod m = \left(x^k - \sum_{i=1}^{k} a_i x^{k-i}\right) \bmod m, \quad (3)$$

with coefficients a_i in $GF(m)$, is a primitive polynomial modulo m, which means that all powers of x modulo $f(x)$ and modulo m constitute S^*.

Let $r = (m^k - 1)/(m - 1)$. The following conditions are necessary and sufficient for $f(x)$ to be primitive modulo m [52]:

(a) $((-1)^{k+1}a_k)^{(m-1)/q} \bmod m \neq 1$ for each prime factor q of $m-1$;
(b) $((x^r \bmod f(x)) \bmod m) = ((-1)^{k+1}a_k) \bmod m$;
(c) $((x^{r/q} \bmod f(x)) \bmod m)$ has degree >0 for each prime factor q of r, $1 < q < r$. ■

For large values of m^k, factorizing r is often very difficult. It becomes the bottleneck in checking the above conditions [57, 58]. It is a good idea then to seek couples (m, k) such that r is prime, since checking primality is much easier than factoring [71]. Given m, k and the factorizations of $m-1$ and r, it is relatively easy to find primitive polynomials simply by random search for proper a_i's. For prime m, there are exactly

$$N(m, k) = (m^k - 1)(1 - 1/q_1)(1 - 1/q_2)\cdots(1 - 1/q_h)/k$$

choices of $(a_1, \ldots, a_k)$ that satisfy the above sufficient conditions, where $q_1, \ldots, q_h$ are the distinct prime factors of $m^k - 1$ [52]. In the case $k = 1$, a primitive polynomial $x - a_1$ means that a_1 is a primitive element modulo m, and whenever one such a_1 has been found, all others can be found easily, since they are exactly all the integers of the form $a_1^j \bmod m$ where j is relatively prime to $m-1$.

For the maximal period to be attained, A must be nonsingular in arithmetic modulo m, since otherwise $AX \bmod m = 0$ for some vector $X \neq 0$. Then, if A^{-1} denotes the inverse of A, we have $X_{n-1} = A^{-1} X_n \bmod m$, so that the sequence can be generated in reverse order. The matrix $A^{-1} = A^{p-1} \bmod m$ can be computed using "divide-to-conquer" as we will see later.

Composite Modulus

When m is not prime and $C = 0$, the maximal period is typically much smaller than m^k. For $m = p^e$, p prime and $e \geq 1$, the maximal possible period is $(p^k - 1)p^{e-1}$, except for $p = 2$ and $k = 1$, where it is 2^{e-2} [33, 52]. Sufficient conditions under which this period is attained and a simple method for constructing matrices A giving maximal period generators are given in [33]. The exception $p - 1 = k = 1$ is treated in [52]. The case where $p = 2$ has some interest in terms of implementation, but the cost in terms of period length, for a given approximate size of m, is important. For example, for $p = 2$ and $k = 1$, the maximal period is $m/4$, while it is $m - 1$ for prime m. For $m = 2^{31}$ and $k = 5$, the longest possible period is $(2^5 - 1)2^{31-1} = 2^{35} - 2^{30}$, while $m^k - 1 = 2^{155} - 1$ is about 2^{120} times longer! This is one reason why it is often recommended that only prime values of m be used. There are also other important reasons. A major one is that for small p, the low order bits do not look random at all. For $p = 2$ and $k = 1$, the i-th least significant bit of X_n has period equal to $\max(1, 2^{i-2})$ [13, 24]. If the period of such a generator is split into 2^d equal segments, then all segments are identical except for their d most significant bits [24, 28]. For $i = 2^{e-d-2} > 0$, all points (x_n, x_{n+i}) lie on at most $\max(2, 2^{d-1})$ parallel lines [24]. For $k > 1$ (still with $p = 2$), the maximal period for the d-th least significant bit is $(2^k - 1)2^{d-1}$.

With $C \neq 0$, for $k = 1$, it is possible to obtain a period length of m. Conditions are given in Knuth [52]. For $p = 2$ and $k = 1$, the period of the i-th least significant bit of X_n is at most 2^i and the pairs (x_n, x_{n+i}), for $i = 2^{e-d}$, lie in at most $\max(2, 2^{d-1})$ parallel lines [24]. For $k > 1$, since any k-th order LCG is equivalent to some $(k+1)$-th order MLCG, a general upper bound on the period length is $(p^{k+1} - 1)p^{e-1}$. Again, for large e and k, this is much smaller than m^k.

Jumping Ahead, Splitting, and Vectorization

Jumping ahead in the sequence of a MLCG can be done efficiently using

$$X_{n+j} = (A^j X_n) \bmod m = (A^j \bmod m) X_n \bmod m.$$

The matrix $(A^j \bmod m)$ can be precomputed using the divide-to-conquer algorithm [11]:

$$A^j \bmod m = \begin{cases} A & \text{if } j = 1; \\ A \times A^{j-1} \bmod m & \text{if } j > 2, j \text{ odd}; \\ A^{j/2} \times A^{j/2} \bmod m & \text{if } j > 1, j \text{ even}. \end{cases}$$

Such "jumping ahead" facilities are required for *splitting* the sequence into long disjoint subsequences. This is useful for many simulation applications [13, 28, 59].

On parallel computers, *vectorization* techniques can be used to generate many subsequences simultaneously [16, 24, 53]. Given J processors, one can precompute $A^J \bmod m$ and use it as a multiplier on all the processors, starting with seed X_{j-1} on processor j. This way, each processor generates values that are J positions apart in the basic sequence. A second approach is to use multiplier $A^j \bmod m$ on processor j, with a common seed on all the processors, and use the "new state" of processor J as a seed for the next step. This way, the successive seeds are J values apart in the basic sequence and the processors generate exactly the same values as in the first approach. One drawback is that all processors must have access to the state of processor J. A third approach which we recommend, called *splitting*, is to keep multiplier A on each processor, but to start with different seeds that are far apart in the basic sequence. This is more appealing in practice, since it does not change the multiplier A whose choice, typically, is dictated by ease of implementation criteria. The first approach is in fact equivalent to splitting, but combined with a change of multiplier. To generate the (far apart) seeds, we use multiplier $A^\nu \bmod m$ for some *huge* value of ν, often a power of two (but beware if m is itself a power of two; see following). New seeds could be computed only as needed. Implementation with this "jumping" multiplier could be more complicated and much slower than for A, but it is used only to produce the seeds. (Also see [59].)

One concern with splitting is that long-range correlations become important. Vectors formed by output values from different substreams should be well distributed in the unit hypercube. For instance, if seeds are spaced ν values apart, we might have special interest for lag ν correlation. As mentioned, if m and ν are powers of two, the substreams are identical except for their (few) most significant bits. Further, for c such that 2^c is smaller than the number of substreams, each substream has 2^c *companion* substreams that differ only in their c most significant bits. Clearly, in this case, ν should not be equal to (or near) a power of two. Durst [28] suggests choosing seeds randomly, after comparing that to regular spacing with $\nu = 1{,}000{,}001$ for $m = 2^{48}$. For prime m, $k = 1$, and full period, all pairs of the form $(x_n, x_{n+(m-1)/2})$ lie on a line with slope -1 [24].

Another approach, suggested for instance in [49, 86], is to use different additive constants C (and the same A and m) for the different substreams. But in fact, as mentioned by Durst [28], changing the constant does not really change the generator. As we will see below, the multidimensional lattice structure of a LCG is independant of C (except for some shifting). Also, consider the LCG (1) and let

$$Y_n = (X_n - D) \bmod m \qquad (4)$$

for some constant vector D. Then, one has

$$Y_{n+1} = (AY_n + C + (A - I)D) \bmod m \qquad (5)$$

where I is the identity matrix. That is, all generators with additive constant of the form $C + (A - I)D$ for $D \in S$ produce the same sequence as (1), except for a shift of $-D$, modulo m. For maximal period generators with $k = 1$ for which $a \bmod 8 = 5$, there are only two such sequences (one for $c = 1$ and one for $c = 3$) and they are in fact antithetical.

Of course, one can use completely different generators on the different processors (or for the different substreams), or simply different multipliers. This appears more troublesome in terms of management. However, finding millions of good generators is not really a problem [28].

Implementations

Implementing (1) in a portable way, in high-level language, is tricky in general, because m is typically near the largest integer representable on the machine and the products involved in computing (1) will overflow. We now discuss ways of computing $ax \bmod m$ for integers a and x.

If $m = 2^e$ where e is the number of bits on the computer word, and if one can use unsigned integers without overflow checking, the products modulo m are easy to compute: just discard the overflow. This is quick and simple. For that reason, MLCGs with moduli of this form are used abundantly in practice, despite their serious drawbacks. Some nuclear physicists, for instance, perform simulations that use billions of random numbers on supercomputers and are quite reluctant to give up using them [28, 49]. Usually, they also generate many substreams in parallel. In view of the above remarks, all this appears dangerous. Perhaps some people like playing with fire.

For more general m, representable as an integer on the target computer, [13, 56, 85] give an efficient and easily implementable way to compute $ax \bmod m$, for $0 < x < m$, when

$$a(m \bmod a) < m. \qquad (6)$$

If we decompose $m = aq + r$ where $r < a$, that condition becomes $r < q$, in which case $a = (m - r)/q = \lfloor m/q \rfloor$. It is then easy to see that all multipliers a satisfying $r < q$ are of the form $a = i$ or $a = \lfloor m/i \rfloor$ for $i < \sqrt{m}$. Note that in view of this condition, it might be worthwhile considering negative multipliers a. Using $a < 0$

On parallel computers, vectorization techniques can be used to generate many subsequences simultaneously.

is equivalent to using $a + m$, but condition (6) might hold for $-a$ and not for $a + m$. In the following Pascal-like code to perform $x := ax \bmod m$, if $r < q$, all values during the computations will remain between $-m$ and m.

```
k := x DIV q;
x := a * (x - k * q) - k * r;
IF x < 0 THEN x := x + m
```

For small a, another approach is to perform the computations in double-precision floating-point [56]. This could be faster on some computers with floating-point coprocessors. Carta [15] describes a different low-level implementation technique for $m = 2^{b-1} - 1$ on b-bit machines. The smaller is a, the faster it goes (in average). He also introduces a faster "alternative algorithm" which actually changes the generator. We believe that this is dangerous and should be avoided. Techniques for computing $ax \bmod m$ in a high-level language for the more general case are studied in [59], which also gives portable codes.

Multiple Recursive Generators

For a given prime m, whether a MLCG has full period or not depends only on the characteristic polynomial of its matrix. Any polynomial of the form (3) has a *companion* matrix

$$A = \begin{pmatrix} 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ a_k & a_{k-1} & \cdots & a_1 \end{pmatrix} \quad (7)$$

whose $f(x)$ is the characteristic polynomial. When the matrix A has this special structure, the first $k - 1$ components of X_n are obtained by shifting the last $k - 1$ components of X_{n-1}, and the last component of X_n is a linear combination of the components of X_{n-1}. This can be viewed as producing a sequence of integers, each one defined as a linear combination modulo m of the k previous ones. This kind of generator is called *multiple recursive* (MRG) [46, 52]. With a matrix of this form, and denoting

$$X_n = (x_n, \ldots, x_{n+k-1})', \quad (8)$$

equation (2) is equivalent to the recursion

$$x_n := (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod m. \quad (9)$$

Restricting our search to generators of this class is certainly supported by their ease of implementation. It is further reinforced by the following property [45, 58, 76]: for any generator defined by (2), with the characteristic polynomial of A defined by (3), the sequence of states obeys the recursion

$$X_n := (a_1 X_{n-1} + \cdots + a_k X_{n-k}) \bmod m. \quad (10)$$

In other words, each component of X_n evolves according to the same recursion (9), which means that we just have k copies of the same MRG evolving in parallel (hopefully, with different and "far apart" seeds). This gives a good argument supporting the direct use of (9).

Another interesting special case in terms of implementation is when the characteristic polynomial $f(x)$ is only a trinomial, of the form $f(x) = x^k - a_j x^{k-j} - a_k$, for $1 \le j < k$. Primitive trinomials of this form are easy to find [57, 58]. The corresponding recursion becomes:

$$x_n := (a_j x_{n-j} + a_k x_{n-k}) \bmod m. \quad (11)$$

The generator can be implemented directly in this form, with its state redefined as the vector $(x_{n-1}, \ldots, x_{n-k})$. The "vectorized" recursion (10) becomes

$$X_n := (a_j X_{n-j} + a_k X_{n-k}) \bmod m. \quad (12)$$

The state then becomes the matrix $s_n = (X_{n-1}, \ldots, X_{n-k})$. It can have interest for parallel computers (Also see section on GFSR and Lagged-Fibonacci generators)

Lattice Structure and Spectral Test

Consider a maximal period MRG, of the form (9), and let

$$T_t = \{(x_n, \ldots, x_{n+t-1}),\ n \ge 0\} \cup \{0\}$$

be the set of all *overlapping* t-tuples of successive values, plus the zero vector. It is well known [26, 45, 46, 52, 57, 58, 64] that the periodic continuation of T_t with period m,

$$L_t = T_t + m\mathbb{Z}^t,$$

forms a lattice with unit cell volume of $\max(1, m^{t-k})$. Recall that a t-dimensional lattice is a set of the form

$$L = \left\{ \sum_{i=1}^{t} z_i V_i, \text{ each } z_i \text{ integer} \right\}$$

where $V_1, \ldots, V_t$ is a set of linearly independent vectors called a basis. A set of vectors $W_1, \ldots, W_t$ such that the scalar products obey $V_i \cdot W_j = \delta_{ij}$ form a basis of the *dual* lattice. Bases for L_t and its dual can be constructed easily as explained in [45, 48].

For $t \le k$, the lattice L_t contains all possible integer vectors and the unit cell volume is one. For $t = k$, each vector except the zero vector occurs once and only once over the period. For $t > k$, the unit cell volume can be huge compared to 1, which is the value that one would expect from truly random integer vectors. This can be viewed as a strong limitation of simple LCGs (with $k = 1$) and suggests using large values of k. A unit cell of the lattice is determined by the vectors of a *Minkowski-reduced lattice base* (MRLB) [2, 3, 45]. It is traditionally accepted that "better" generators are obtained when the unit cells of the lattice are more "cubic-like" (i.e. when the vectors of the MRLB have about the same size). The ratio q_t of the sizes of the shortest and longest vectors of a MRLB is called the *Beyer-quotient*. It can be used to assess the quality of the lattice. Values near one are said to be more desir-

able. Note however that reducing the unit cell volume (by increasing m, or k, or both) can be much more effective in improving the quality than getting a larger Beyer-quotient with fixed m and k. Afflerbach and Grothe [2, 45] give efficient algorithms to compute a MRLB and the Beyer-quotient for a given lattice. A figure of merit can be $Q_T = \min_{k<t\leq T} q_t$ for some large enough T.

The lattice structure also means that all points of T_t lie in a family of equidistant parallel hyperplanes. Among all such families of hyperplanes that cover all the points, choose the one for which the successive hyperplanes are farthest apart, and let D_t be the distance between them. The smaller that distance, the better, since this implies thinner empty "slices" in the lattice. Dieter [26] (see also [52]) gives an algorithm to compute D_t, which is in fact equal to one over the length of the shortest vector in the *dual lattice* to L_t. This shortest vector can also be computed using the algorithms of [2, 45], which are faster for large t. For given m and k, the number of hyperplanes in the chosen family cannot exceed $(t!(m^k - 1))^{1/t}$ and there is also a theoretical lower bound D_t^* on D_t [36, 52, 56]. One can define the figures of merit $S_t = D_t^*/D_t$ and $M_T = \min_{k\leq t\leq T} S_t$, which lie between 0 and 1. For $k = 1$ and using M_6 as a criterion, computer searchs to find good generators have been done by Fishman and Moore [36] (for $m = 2^{31} - 1$), by Fishman [35] (for $m = 2^{32}$ and $m = 2^{48}$), by L'Ecuyer [56] (for different values of m near 2^{31} and $a_1^2 < m$), and by Park and Miller [85] (for $m = 2^{31} - 1$ and a_1 (m mod a_1) $< m$). L'Ecuyer and Blouin have performed more extensive searchs, for $1 \leq k \leq 7$ and different values of m up to near 2^{63}, first using M_8 as a criterion [57], then using Q_{20} [58].

The results of [58] show that for $k > 2$, generators of the special form (11) have Beyer quotients much smaller than 1. But these generators are faster than those of the general form. In fact, for a given generator of the general form (9) with $k > 2$, one can usually find a generator of order $k' > k$, of the special form (11), that will be faster and will have smaller D_t for all $t > k$. Its Beyer quotients might be smaller, but this is compensated for by much smaller unit cell volumes. In that case, D_t appears to be a better "absolute" criterion for comparing generators with different values of k and m.

Table I gives a few values. For the first 3 columns, all multipliers a_i satisfy condition (6): a_i (m mod a_i) $< m$. (The last column will be discussed later.) For $m = 2^{31} - 1$ and $k = 1$, the multiplier given is the one with the largest Q_{20} among all those that satisfy this condition. Those in columns 2 and 3 (for $k > 1$) were obtained by extensive random search and are believed to be close to the best ones with respect to Q_{20}. For comparison, for $m = 2^{31} - 1$ and $k = 1$, the multiplier $a = 742938285$ recommended Fishman and Moore [36] has $Q_{20} = .5808$, while for $a = 16807$, $a = 48271$ and $a = 69621$, which are mentioned in Park and Miller [85] and satisfy (6), the respective values are .1315, .4563 and .5373.

A similar lattice structure appears when all components of X_n are used at each iteration [3, 44, 45]. It can be analyzed in a similar way. When the generator is *not* multiplicative ($C \neq 0$), the lattice is shifted by a constant vector, yield-

TABLE I. Lattice Properties of Some MLCGs

m	$2^{31} - 1$	$2^{31} - 1$	$2^{31} - 19$	4611685301167870637
k	1	2	7	1
a_1	41358	46325	1103197117	1968402271571654650
a_2		1084587	0	
a_3			0	
a_4			0	
a_5			0	
a_6			0	
a_7			1319713409	
Q_{20}	0.6209	0.3575	0.0000436	0.1443
$Q_{20} =$	q_5	q_3	q_{12}	q_4
$1/m$	4.65E-10	4.65E-10	4.65E-10	2.17E-19
d_2	2.417E-5			6.50E-10
d_3	9.811E-4	9.211E-7		7.002E-7
d_4	5.572E-3	2.578E-5		4.635E-5
d_5	1.507E-2	2.422E-4		2.008E-4
d_6	3.242E-2	8.143E-4		8.890E-4
d_7	5.025E-2	2.346E-3		2.621E-3
d_8	8.006E-2	5.011E-3	6.996E-7	5.782E-3
d_9	.1031	9.372E-3	4.929E-6	9.571E-3
d_{10}	.1054	1.856E-2	5.249E-6	1.738E-2
d_{11}	.1690	1.856E-2	6.807E-6	2.361E-2
d_{12}	.1690	2.588E-2	8.469E-6	3.077E-2
d_{13}	.1767	3.483E-2	9.743E-6	3.477E-2
d_{14}	.1767	4.291E-2	2.000E-5	3.968E-2
d_{15}	.2425	5.177E-2	3.684E-5	5.987E-2
d_{16}	.2425	6.509E-2	7.755E-5	6.074E-2
d_{17}	.2672	6.835E-2	1.271E-4	6.509E-2
d_{18}	.2672	7.392E-2	2.019E-4	7.433E-2
d_{19}	.2672	8.362E-2	2.871E-4	8.192E-2
d_{20}	.2773	9.449E-2	4.185E-4	8.771E-2

ing what is called a *grid*. The corresponding lattice can be analyzed in the same way, since its structure does not depend on C. When T_t is replaced by the set of *non-overlapping* t-tuples, L_t does not form a lattice in general [1].

Tausworthe, GFSR, Lagged-Fibonacci

When $a_j = |a_k| = 1$, the recursion (12) is a special case of the so-called *lagged-Fibonacci* generator (LFG) [65, 66]. A LFG is defined by

$$X_n := (X_{n-j} \diamond X_{n-k}) \bmod m \quad (13)$$

where $\diamond$ is any componentwise binary operation (sum, product, subtraction, etc.) and X_n is a vector of any size, with components in $\{0, \ldots, m-1\}$. These generators are analyzed in [65, 66], for different operators $\diamond$, and $m = 2^e$. For such values of m, their maximal period lengths are typically much smaller than $m^k - 1$.

Increasing the period of a LCG can be done by taking a larger m or a larger k. Typical MLCGs use m near 2^{31} and small k. An opposite extreme is to take $m = 2$, with large k. In this case, X_n is a vector of k bits. For $b \le k$, one can interpret, say, the last b bits of X_n as a b-bit integer. The generator thus obtained is called *Generalized Feedback Shift Register* (GFSR) [37, 38, 39, 40, 41, 42, 62, 66, 93, 94, 95, 96]. For the "efficient" special case (12), with $a_j = a_k = 1$, it becomes a special kind of lagged-Fibonacci generator, with operator $\diamond = \oplus$ (the bitwise exclusive or). Since the first $k - b$ bits (if any) are unused, X_n can be viewed as a b-bit vector. The generator's state s_n is a $b \times k$ matrix of bits. Recommended values are for example $b = 31$ and $k = 521$ or 607 [37].

For $b = 1$, one gets a MRG which produces a sequence of bits (also called a M-sequence, for maximal period generators). Tausworthe [92] suggested regrouping blocks of successive bits to form integers or reals. These *Tausworthe* (or simple shift register) generators are rather slow and are almost not used anymore in practice. But see also [96]. GFSR generators are faster but use more memory. Since a GFSR generator corresponds in fact to b copies of the same M-sequence evolving in parallel, one should use "jumping ahead" techniques to compute an initial matrix of bits so that these b bit-generators have their seeds far enough apart. (This also applies to LFGs in general.) Initialization procedures have been proposed (e.g., in [8, 21, 42]). But Fushimi [39] gives a much simpler and faster procedure, which guarantees equidistributivity in all dimensions $t \le \lfloor k/b \rfloor$ and good autocorrelation properties for lags up to $\lfloor (2^k - 1 - b)/b \rfloor$. The basic idea is to find a Tausworthe generator that is *equivalent* to the target GFSR and use the former to compute an initial matrix of bits.

Marsaglia [65, 66] argues against the use of GFSR generators. He describes a statistical test, based on the ranks of random binary matrices, that some GFSR generators fail. But in fact, such specific tests to "catch-up" generators of a given class can be built for most classes of generators currently in use. Recent studies [7, 38, 41, 77, 93, 94, 95] indicate that GFSRs with large order and properly chosen parameters have excellent statistical properties in general. One problem, though, is that they use a large amount of memory. This is particularly true when many generators have to be run in parallel. A better idea could be to stay away from the two extreme cases $k = 1$ and $m = 2$. Pick a large but practical m and increase k as needed.

Other Variants

The ACORN generator proposed recently in [101] is in fact equivalent to a MLCG with matrix A such that $a_{ij} = 1$ for $i \ge j$, $a_{ij} = 0$ otherwise. Generators based on *cellular automata* are discussed in [22, 98] and other references given there. The generators proposed by Tindo [98] are equivalent to LCGs where the constant C is a vector of ones, while the matrix A has identical elements a_0 on its diagonal, identical elements a_1 on its subdiagonal, and zeros elsewhere. The maximal possible period is $m^k - m$ and finding generators that reach it is relatively easy. The elements of the vectors X_n are combined to produce the output.

Non-Linear Generators

LCGs can be generalized to *quadratic* generators of the form

$$X_n := (X'_{n-1} A X_{n-1} + B X_{n-1} + C) \bmod m,$$

where A and B are $k \times k$ matrices, or more generally to

$$X_n := P(X_{n-1}) \bmod m$$

where P is some multivariate polynomial. See [74]. For $k = 1$, quadratic generators are analyzed in [52, 30]. The latter authors show that for maximal period generators ($\rho = m$), the *non-overlapping* t-tuples determine a union of grids (shifted lattices).

A class of generators based on Tchebychev mixing are known to have bad statistical properties [50]. Classes of LCGs with randomly varying multipliers and/or additive constants are discussed in [17, 19]. They have interesting theoretical properties, but they require truly random bits at each step.

A Class of Generators by Inversion

Eichenauer et al. [29, 31] introduced a class of "non-linear" generators using a sequence $\{x_n,\ n \ge 0\}$ that obeys (9) for prime m. Let $\tilde{x}_i$ be the i-th non-zero value x_n in that sequence. Define $z_n = (\tilde{x}_{n+1}\tilde{x}_n^{-1}) \bmod m$, where $\tilde{x}_n^{-1}$ denotes the inverse element of $\tilde{x}_n$ in $GF(m)$. The z_n's are then used to produce the u_n's. A version of Euclid's algorithm, whose average running time is approximately $12(\ln 2)(\ln m)/\pi$ [52], can be used to compute the inverse $\tilde{x}_n^{-1}$. Divide-to-conquer can also be used as mentioned previously, since jumping back one value is equivalent to jumping ahead ρ-1

values. Computing this inverse in software on a standard computer is slow, which makes these generators somewhat inefficient. However, if implemented in hardware, the inversion procedure can be practically as fast as an ordinary floating-point division [29, and personal communication from D. E. Knuth to J. Lehn].

For prime m, the maximal possible period length for the z_n's is m^{k-1}. Sufficient conditions for it to be attained are given in [31]. Maximal period generators are easy to find. For $k = 2$ or 3, one can write a recursion directly for the z_n's. For $k = 2$, it is

$$z_n = \begin{cases} (a_1 + a_2 z_{n-1}^{-1}) \bmod m & \text{if } z_{n-1} \neq 0; \\ a_1 & \text{if } z_{n-1} = 0. \end{cases}$$

The main motivation behind these generators is that the sequence they produce does not share the lattice property of the usual LCGs. Their structure is highly non-linear: any t-dimensional hyperplane contains at most t overlapping t-tuples of successive values [34, 78, 79, 84]. Niederreiter [82] shows that they behave very much like truly random generators with respect to discrepancy. Therefore, their theoretical properties look quite good.

These non-linear generators can also be viewed as a way of implementing $g : S \rightarrow U$ for a LCG, (i.e. as a supplementary step when transforming the state of the LCG into a value between 0 and 1). Other ways of defining g will be discussed later.

Combined Generators

To increase the period and try to get rid of the regular patterns displayed by LCGs, it has often been suggested that different generators be combined to produce a "hybrid" one [20, 47, 48, 52, 56, 65, 67, 73, 99, 100]. Such combination is often viewed as completely heuristic and is sometimes discouraged. Ripley [90], for instance, views it as "better the unknown than the devil we know" attitude. But besides being strongly supported by empirical investigations, combination has some theoretical support. First, in most cases, the period of the hybrid is much longer than that of each of its components, and can be computed. Second, there are theoretical results suggesting that some forms of combined generators generally have better statistical behavior. For instance, suppose two random sequences $\{x_n, n \geq 0\}$ and $\{y_n, n \geq 0\}$ are combined elementwise to form a third sequence $\{z_n, n \geq 0\}$, where $z_n = x_n \diamond y_n$ and $\diamond$ denotes some binary operator. Assume the three sequences are defined over the same finite set. Then, under fairly reasonable conditions, the t-tuples of successive values are "more" (or at least as much) uniformly distributed in some sense for the third sequence than for any of its two constituents. See [14, 65]. Recall, however, that the generators used in practice produce completely deterministic sequences. In that context, the above results might raise optimism, but give no guarantee of quality. As pointed out in [13], combination can conceivably worsen things. Niederreiter (personal communication) points out that if x_n and y_n have inverses with respect to $\diamond$, which is often the case in deterministic settings, then $x_n = z_n \diamond y_n^{-1}$ and $y_n = x_n^{-1} \diamond z_n$, and the same argument as above suggests that x_n and y_n have "better" statistical properties than z_n!

Some combination approaches are based on *shuffling* [13, 52, 54, 73]. In one of the variants, two simple generators are used, one to fill the cells of a buffer and the other to select which cell the next output value will be taken from. At each step, the second generator selects a cell, outputs its content, then the first generator fills it back. Shuffling is not so well understood and has some practical drawbacks [13]. For instance, there is no obvious efficient way to jump ahead in the sequence.

L'Ecuyer [56] proposed a combination method for MLCGs of order $k = 1$ with distinct prime moduli $m_1, \ldots, m_J$. If x_{jn} denotes the state of generator j at step n, define the combination (slightly more general than in [56]):

$$Z_n = \left(\sum_{j=1}^{J} \delta_j x_{jn} \right) \bmod m_1 \quad (14)$$

for some integers δ_j. In [56], $\delta_j = (-1)^{j-1}$ is suggested. This is related to the following generalization of the combination approach proposed by Wichmann and Hill [99], which is a bit slower because it requires more divisions:

$$U_n = \left(\sum_{j=1}^{J} \delta_j x_{jn}/m_j \right) \bmod 1. \quad (15)$$

If each individual MLCG has full period $m_j - 1$, then the period of the latter is always equal to the least common multiple of $m_1 - 1, \ldots, m_J - 1$ [61].

It turns out [61, 97] that there exists a MLCG with modulus $m = \prod_{j=1}^{J} m_j$ whose lattice structure approximates quite well the behavior of (14) in higher dimensions, and which is exactly equivalent to (15). This MLCG does not depend on the δ_j's. The equivalence of the Wichman and Hill generator to a MLCG was already pointed out by Zeisel [99]. Such structural properties are not so deceptive as it might appear. In fact, it shows that these combinations can be viewed as efficient ways of implementing (sometimes with added noise) generators with moduli much larger than the largest integer representable on the target computer. However, these large moduli are not prime.

One generator suggested in [56] had $J = 2$, $m_1 = 2147483563$, $m_2 = 2147483399$, $a_1 = 40014$, $a_2 = 40692$, $\delta_1 = 1$ and $\delta_2 = -1$. Its "ap-

proximating" MLCG has $m = 4611685301167870637$ and $a = 1968402271571654650$ (see Table I) [61, 97]. The approximation is quite good in dimensions $t \geq 3$. Figures of merit for the lattice associated with this MLCG are given in the last column of Table 1. They show that for $t \geq 3$ (where the approximation is good), the combined generator has a better structure than the best MLCG of order one (and modulus $m = 2^{31} - 1$), and is quite comparable to the best MLCGs of order 2. In two dimensions, the combined generator is also more "noisy" than these MLCGs. Before suggesting that generator, the author had been unable to detect graphically, with reasonable computer time, any two-dimensional structural property. For the same size, one can also find better combined generators than this one. See [61].

Transforming into U(0,1) Variates

There are different ways of using the state vector $X_n = (x_{n1}, \ldots, x_{nk})'$ of a LCG to produce real values between 0 and 1, that is of defining $g : S \rightarrow U$ where U is some finite subset of [0,1]. When m is large, a component x_{ni} can be simply divided by m, yielding a result in [0,1). But it is often necessary to make sure that the result lies *strictly* between 0 and 1. This can be accomplished by dividing instead by $m + 1$, replacing first x_{ni} by m when $x_{ni} = 0$. Other slightly more involved techniques are proposed in [68, 70].

Afflerbach and Grothe [3, 44, 45] use *all* the components of X_n to obtain k $U(0,1)$ variates at each iteration. L'Ecuyer and Blouin [57, 58] use only x_{nk} (the last component), which is equivalent to using a MRG. As discussed previously, the former is equivalent to splitting.

When dividing x_{ni} by $(m + 1)$, the *mesh size* (or "granularity") of the output is $1/(m + 1)$. For some applications, a smaller mesh size might be necessary (see, e.g., [29, 88]). One can then use a *digital method*, in which a value $u_n \in (0,1)$ is produced by

$$u_n = \left(\sum_{i=1}^{t} p^{-i} x_{tn+i-1} \right) \bmod (1 - p^{-t}),$$

where $p \leq m + 1$ and $t \geq 1$ are integers (p could be for instance a power of two), and $\{x_i, i \geq 0\}$ is the sequence of all used vectors components (or a sequence produced by (9)). Other variants are discussed in [72, 76, 80]. Tausworthe and GFSR generators use a similar technique with $p = 2$. In the MRG case, the period of the u_n's always divides $m^k - 1$. When $m^k - 1$ and t are relatively prime, it is almost always $m^k - 1$.

Statistical Testing

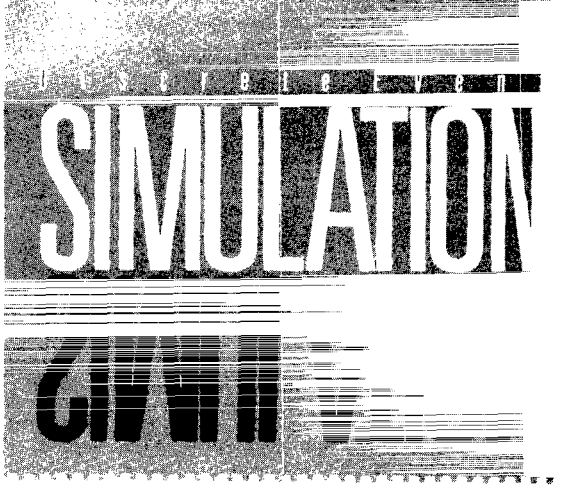
Knuth [52] describes a set of empirical statistical tests, usually viewed as the "standard" ones. Many of them are included in the package of Dudewicz and Ralley [27]. Marsaglia [65] describes others, supposedly more powerful. Statistical tests are rather easy to design: any function of a finite set of i.i.d. uniform random variables can be used as a statistic to define a test of hypothesis, if its distribution is known. To gain power, the test can be repeated N times, and the empirical distribution of the values of the statistic can be compared to its theoretical distribution, using, for instance, the Kolmogorov-Smirnov test [27, 56]. Of course, the quality of a generator can never be *proven* by any statistical test.

Discrepancy

Besides empirical tests, some theoretical tests can give information about the statistical behavior of certain generators, often over the full period but sometimes also for just part of the period. Examining the lattice structure of LCGs yields such tests. Other tests are based on the notion of *discrepancy*. Informally, the discrepancy $D_N^{(t)}$ in t dimensions is the absolute difference between the expected number and actual number of vectors $(x_n, \ldots, x_{n+t-1})$, $0 \leq n < N$, falling into a hyperrectangular region with sides parallel to the axes, maximized over all such regions (or in some definitions, over those regions with a corner at the origin). Intuitively, a discrepancy that is "too high" should be avoided. Also, a discrepancy that is "too low" can indicate a sequence that is "too regular". Some "very regular" (so-called *quasirandom*) sequences, whose discrepancy has an order of magnitude lower than that of genuinely random sequences, are useful for some applications [12, 75, 84]. For many different classes of generators, bounds on $D_N^{(t)}$ are available [72, 75, 76, 77, 80, 81, 82, 84, 94, 95]. These bounds can give some sort of "protection". But only in rare cases, exact values can be computed. For instance, two-dimensional discrepancy can be computed efficiently for a class of LCGs [5]. As pointed out in [4, 52], the discrepancy is very sensitive to rotations of the axis, in contrast to the Beyer-quotient or spectral test. This suggests that rating generators on the basis of their discrepancy bounds is not necessarily the best idea. On the other hand, discrepancy is a useful measure for getting error bounds in numerical integration or for random search procedures. Further, bounds on some statistical quantities such as serial correlation can be obtained in terms of bounds on the discrepancy. Niederreiter's survey [84] puts more emphasis on discrepancy and quasirandom sequences.

Conclusion

A lot has been written on uniform variate generation, but certainly, the last word has not been said. As computing power gets progressively cheaper, applications will require increasingly robust generators. Classical LCGs of order 1 are becoming unsatisfactory for some applications. For example, my laptop computer needs less than 6 hours to loop around the whole period of a MLCG with modulus $m = 2^{32}$ (and period length 2^{30}).



Supercomputers do the same in a few seconds. Increasing the modulus leads to implementation problems. At the other extreme, GFSR generators, which use modulus 2, can attain much longer periods and good statistical properties by using a large order k. However, they use more space. But why stick to these two extreme cases?

MRGs with a trinomial characteristic function, large m and say $k \geq 5$, appear to be an excellent choice in terms of efficiency and statistical quality. The unit cell volume of the associated t-dimensional lattice, for $t > k$, can be reduced by increasing k. The mesh size can be reduced without increasing m by using a digital method to produce the output. Note that the digital method can be implemented using different MRGs (evolving in parallel) for different digits. Non-linear transformations can also be used at this stage, but at the expense of reduced speed if no hardware implementation is at hand. PT-perfect generators offer a good stimulus for further research.

Acknowledgments

This work was performed while the author was with the Département d'informatique, Université Laval, Québec. Comments and suggestions by R. Couturc, U. Dieter, B. L. Fox, M. Fushimi, J. Lehn, H. Niederreiter, G. Perron, S. Tezuka, and the anonymous referee, led to significant improvements. ◻

References

1. Afflerbach, L. The sub-lattice structure of linear congruential random number generators. *Manusc. Math. 55* (1986), 455–465.
2. Afflerbach, L. and Grothe, H. Calculation of Minkowski-reduced lattice bases. *Computing 35* (1985), 269–276.
3. Afflerbach, L. and Grothe, H. The lattice structure of pseudo-random vectors generated by matrix generators. *J. of Comput. and Applied Math. 23* (1988), 127–131.
4. Afflerbach, L. and Weilbächer, R. On Using Discrepancy for the Assessment of Pseudorandom Number Generators. Submitted for publication, 1988.
5. Afflerbach, L. and Weilbächer, R. The exact determination of rectangle discrepancy for linear congruential pseudorandom numbers. *Math. of Comput. 53,* 187 (July 1989) 343–354.
6. Alexi, W., Chor, B., Goldreich, O. and Schnorr, C. P. RSA and Rabin Functions: Certain parts are as hard as the whole. *SIAM J. on Comput. 17,* 2 (1988) 194–209.
7. André, D. L., Mullen, G. L., and Niederreiter, H. Figures of merit for digital multistep pseudorandom numbers. *Math. of Comput.* To be published, 1990.
8. Arvillias, A. C. and Maritsas, D. G. Partitioning the period of a class of m-sequences and application to pseudorandom number generation. *J. ACM 25,* 4 (1978) 675–686.
9. Blum, L., Blum, M. and Schub, M. A simple unpredictable pseudo-random number generator. *SIAM J. Comput. 15,* 2 (1986) 364–383.
10. Boyar, J. Inferring sequences produced by pseudo-random number generators. *J. ACM 36,* 1 (1989) 129–141.
11. Brassard, G. and Bratley P. *Algorithmics, Theory and Practice.* Prentice-Hall, Englewood Cliffs, N.J., 1988.
12. Bratley, P. and Fox, B. L. Algorithm 659: Implementing Sobol's quasirandom sequence generator. *ACM Trans. on Math. Softw. 14,* 1 (Mar. 1988) 88–100.
13. Bratley, P., Fox, B. L. and Schrage, L. E. *A Guide to Simulation.* 2d ed. Springer-Verlag, New York, 1987.
14. Brown, M. and Solomon, H. On combining pseudorandom number generators. *Ann. Stat. 1* (1979) 691–695.
15. Carta, D. G. Two fast implementations of the "minimal standard" random number generator. *Commun. ACM 33,* 1 (Jan. 1990) 87–88.
16. Celmaster, W. and Moriarty, K. J. M. A method for vectorized random number generators. *J. Comput. Phys. 64* (1986) 271–275.
17. Chassaing, P. An optimal random number generator on $\mathbb{Z}_p$. *Stat. and Prob. Lett. 7* (1989) 307–309.
18. Chor, B. and Goldreich, O. Unbiased bits from sources of weak randomness and probabilistic communication complexity. *SIAM J. Comput. 17,* 2 (1988) 230–261.
19. Chung, F. R. K., Diaconis, P., and Graham, R. L. Random walks arising in random number generation. *Ann. Probab. 15,* 3 (1987) 1148–1165.
20. Collings, B. J. Compound random number generators. *J. Am. Stat. Assoc. 82,* 398 (1987) 525–527.
21. Collings, B. J. and Hembree, G. B. Initializing generalized feedback shift register pseudorandom number generators. *J. ACM 33* (1986) 706–711. A, also in *J. ACM, 35,* 4 (1988) 1001.
22. Compagner, A. and Hoogland, A. Maximum length sequences, cellular automata, and random numbers. *J. Comput. Phys. 71* (1987) 391–428.
23. Dagpunar, J. *Principles of Random Variate Generation.* Oxford University Press, 1988.
24. De Matteis, A. and Pagnutti, S. Parallelization of random number generators and long-range correlations. *Numerische Mathematik 53* (1988) 595–608.
25. Devroye, L. *Non-Uniform Random Variate Generation.* Springer-Verlag, New York, 1986.
26. Dieter, U. How to calculate shortest vectors in a lattice. *Math. Comput., 29,* 131 (1975) 827–833.
27. Dudewicz, E. J. and Ralley, T. G. *The Handbook of Random Number Generation and Testing with TESTRAND Computer Code.* American Sciences Press, Columbus, Ohio, 1981.
28. Durst, M. J. Using linear congruential generators for parallel random number generation. In *Proceedings of the 1989 Winter Simulation Conference.* IEEE Press (1989) pp. 462–466.
29. Eichenauer, J. and Lehn, J. A Nonlinear congruential pseudorandom number generator. *Statistische Hefte, 27* (1986) 315–326.
30. Eichenauer, J. and Lehn, J. On the structure of quadratic congruential sequences. *Manusc. Math., 58* (1987) 129–140.

31. Eichenauer, J., Grothe, H., Lehn, J. and Topuzoğlu, A. A multiple recursive nonlinear congruential pseudorandom number generator. *Manus. Math. 59* (1987) 331–346.
32. Eichenauer, J., Lehn, J. and Topuzoğlu, A. A nonlinear congruential pseudorandom number generator with power of two modulus. *Math. Comput. 51,* 184 (1988) 757–759.
33. Eichenauer-Herrmann, J., Grothe, H. and Lehn, J. On the period length of pseudorandom vector sequences generated by matrix generators. *Math. Comput. 52,* 185 (1989) 145–148.
34. Eichenauer-Herrmann, J. Inversive congruential pseudorandom numbers avoid the planes. *Math. Comput.* (1990). To be published.
35. Fishman, G. S. Multiplicative congruential random number generators with modulus 2^{β}: an exhaustive analysis for $\beta = 32$ and a partial analysis for $\beta = 48$. *Math. Comput. 54,* 189 (Jan 1990) 331–344.
36. Fishman, G. S. and Moore III, L. S. An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$. *SIAM J. Sci. and Stat. Comput. 7,* 1 (1986) 24–45.
37. Fushimi, M. Increasing the orders of equidistribution of the leading bits of the Tausworthe sequence. *Inf. Proc. Lett. 16* (1983) 189–192.
38. Fushimi, M. Designing a uniform random number generator whose subsequences are k-distributed. *SIAM J. Comput. 17,* 1 (1988) 89–99.
39. Fushimi, M. An equivalence relation between Tausworthe and GFSR sequences and applications. *Applied Math. Lett. 2,* 2 (1989) 135–137.
40. Fushimi, M. Random number generation on parallel processors. In *Proceedings of the 1989 Winter Simulation Conference.* IEEE Press (1989) pp. 459–461.
41. Fushimi, M. Random number generation with the recursion $X_t = X_{t-3p} \oplus X_{t-3q}$. In *Comput. and Applied Math.* (1990). To be published.
42. Fushimi, M. and Tezuka, S. The k-distribution of generalized feedback shift register pseudorandom numbers. *Commun. ACM 26,* 7 (1983) 516–523.
43. Goldreich, O., Goldwasser, S. and Micali, S. How to construct random functions. *J. ACM 33,* 4 (1986) 792–807.
44. Grothe, H. Matrix generators for pseudo-random vectors generation. *Stat. Hefte 28* (1987) 233–238.
45. Grothe, H. Matrixgeneratoren zur Erzeugung gleichverteilter Pseudozufallsvektoren (in german). Dissertation (thesis), Tech. Hochschule Darmstadt, Germany, 1988.
46. Grube, A. Mehrfach rekursiverzeugte Pseudo-Zufallszahlen (in german). *Zeitschrift für angewandte Math. und Mechanik 53* (1973) T223–T225.
47. Guinier, D. A fast uniform "Astronomical" random number generator. *ACM SIGSAC Rev. 7,* 1 (Spring 1989), 1–13.
48. Haas, A. The multiple prime random number generator. *ACM Trans. Math. Softw. 13,* 4 (1987) 368–381.
49. Halton, J. H. Pseudo-random trees: Multiple independent sequence generators for parallel and branching computations. *J. Comput. Phys. 84* (1989) 1–56.
50. Hosack, J. M. The use of cebysev mixing to generate pseudorandom numbers. *J. Comput. Phys. 67* (1986) 482–486.
51. Kannan, R., Lenstra, A. K. and Lovász, L. Polynomial factorization and nonrandomness of bits of algebraic and some transcendental numbers. *Math. Comput. 50,* 181 (1988) 235–250.
52. Knuth, D. E. *The Art of Computer Programming* Vol. 2 *Seminumerical Algorithms,* 2d ed. Addison-Wesley, 1981.
53. Koniges, A. E. and Leith, C. E. Parallel processing of random number generation for Monte Carlo turbulence simulation. *J. of Comput. Phys. 81* (1989) 230–235.
54. Law, A. M. and Kelton, W. D. *Simulation Modeling and Analysis,* 2d ed., McGraw-Hill, 1991. To be published.
55. L'Ecuyer, P. A portable random number generator for 16-bit computers. *Modeling and Simulation on Microcomputers 1987.* The Society for Computer Simulation (1987), pp. 45–49.
56. L'Ecuyer, P. Efficient and portable combined random number generators. *Commun. ACM 31,* 6 (1988) 742–749 and 774. Also see the correspondance in *Commun. ACM 32,* 8 (1989) 1019–1024.
57. L'Ecuyer, P. and Blouin, F. Linear congruential generators of order $k > 1$. In *Proceedings of the 1988 Winter Simulation Conference,* IEEE Press, (1988), pp. 432–439.
58. L'Ecuyer, P. and Blouin, F. Multiple Recursive and Matrix Linear Congruential Generators. Submitted for publication, 1990.
59. L'Ecuyer, P. and Côté, S. Implementing a random number package with splitting facilities. *ACM Trans. on Math. Softw.* 1990. To be published.
60. L'Ecuyer, P. and Proulx, R. About Polynomial-Time "Unpredictable" Generators. In *Proceedings of the 1989 Winter Simulation Conference,* IEEE Press, (1989), pp. 467–476.
61. L'Ecuyer, P. and Tezuka, S. Structural Properties for Two Classes of Combined Generators. Submitted for publication, 1990.
62. Lewis, T. G. and Payne, W. H. Generalized feedback shift register pseudorandom number algorithm. *J. ACM 20,* 3 (1973) 456–468.
63. Lidl, R. and Niederreiter, H. *Introduction to Finite Fields and Their Applications.* Cambridge University Press, Cambridge, 1986.
64. Marsaglia, G. Random numbers fall mainly in the planes. In *Proceedings of the National Academy of Sciences of the United States of America 60* (1968) pp. 25–28.
65. Marsaglia, G. A Current View of Random Number Generation. *Computer science and statistics.* In *Proceedings of the Sixteenth Symposium on the Interface* (Atlanta, March 1984). Elsevier Science Publ., North-Holland, 1985, pp. 3–10.
66. Marsaglia, G. and Tsay, L.-H. Matrices and the structure of random number sequences. *Linear Algebra and its Appl. 67* (1985) 147–156.
67. Marsaglia, G., Zaman, A., and Tsang, W. W. Towards a universal random number generator. *Stat. and Prob. Lett. 8* (1990) 35–39.
68. Marse, K. and Roberts, S. D. Implementing a portable FORTRAN uniform (0,1) generator. *Simulation 41,* 4 (1983) 135–139.
69. Modianos, D. T., Scott, R. C. and Cornwell, L. W. Testing intrinsic random number generators. *Byte,*

12, 1 (1987) 175–178.

70. Monahan, J. F. Accuracy in Random Number Generation. *Math. of Comput. 45,* 172 (1985) 559–568.

71. Morain, F. Implementation of the Atkin-Goldwasser-Kilian primality testing algorithm. Rap. de recherche 911, INRIA, Rocquencourt, France, 1988.

72. Mullen, G. L. and Niederreiter, H. Optimal characteristic polynomials for digital multistep pseudorandom numbers. *Computing 39* (1987) 155–163.

73. Nance, R. E. and Overstreet, C., Jr. Some experimental observations on the behavior of composite random number generators. *Oper. Res. 26,* 5 (1978) 915–935.

74. Narkiewicz, W. *Uniform Distribution of Sequences of Integers in Residue Classes.* Lecture Notes in Mathematics, 1087, Springer-Verlag, 1984.

75. Niederreiter, H. Quasi-Monte Carlo methods and pseudorandom numbers. *Bull. Am. Math. Soc. 84,* 6 (1978) 957–1041.

76. Niederreiter, H. A pseudorandom vector generator based on finite field arithmetic. *Math. Japonica 31,* 5 (1986) 759–774.

77. Niederreiter, H. A statistical analysis of generalized feedback shift register pseudorandom number generators. *SIAM J. Sci. Stat. Comput. 8,* 6 (1987) 1035–1051.

78. Niederreiter, H. Remarks on nonlinear congruential pseudorandom numbers. *Metrika 35* (1988) 321–328.

79. Niederreiter, H. Statistical independence of nonlinear congruential pseudorandom numbers. *Monatshefte für Mathematik 106* (1988) 149–159.

80. Niederreiter, H. The Serial Test for Digital *k*-Step Pseudorandom Numbers. *Math. J. Okayama Univ. 30* (1988) 93–119.

81. Niederreiter, H. The serial test for congruential pseudorandom numbers generated by inversions. *Math. of Comput. 52,* 185 (1989) 135–144.

82. Niederreiter, H. Lower bounds for the discrepancy of inversive congruential pseudorandom numbers. *Math. Comput.* 1990. To be published.

83. Niederreiter, H. Statistical independance properties of pseudorandom vectors produced by matrix generators. *J. Comput. Appl. Math.* 1990. To be published.

84. Niederreiter, H. Recent trends in random number and random vector generation. *Ann. Oper. Res.* 1990. To be published.

85. Park, S. K. and Miller, K. W. Random number generators: Good ones are hard to find. *Commun. ACM 31,* 10 (1988) 1192–1201.

86. Percus, D. E. and Kalos, M. Random number generators for MIMD parallel processors. *J. Parallel and Distributed Comput. 6* (1989) 477–497.

87. Reif, J. H. and Tygar, J. D. Efficient parallel pseudorandom number generation. *SIAM J. Comput. 17,* 2 (1988) 404–411.

88. Ripley, B. D. The Lattice Structure of Pseudo-random Number Generators. In *Proceedings of the Royal Society of London, 389 Ser. A,* (1983) pp. 197–204.

89. Ripley, B. D. *Stochastic Simulation.* Wiley, New York, 1987.

90. Ripley, B. D. Uses and abuses of statistical simulation. *Math. Prog., 42* (1988) 53–68.

91. Stern, J. Secret linear congruential generators are not cryptographically secure. In *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science* (1987) pp. 421–426.

92. Tausworthe, R. C. Random numbers generated by linear recurrence modulo two. *Math. of Comput., 19* (1965) 201–209.

93. Tezuka, S. Walsh-spectral test for GFSR pseudorandom numbers. *Commun. ACM 30,* 8 (Aug. 1987) 731–735.

94. Tezuka, S. On the discrepancy of GFSR pseudorandom numbers. *J. ACM 34,* 4 (Oct. 1987), 939–949.

95. Tezuka, S. On optimal GFSR pseudorandom number generators. *Math. of Computat. 50,* 182 (Apr. 1988) 531–533.

96. Tezuka, S. Random number generation based on the polynomial airthmetic modulo two. Rep. RT-0017, IBM Research, Tokyo Research Laboratory, Oct. 1989.

97. Tezuka, S. Analysis of L'Ecuyer's combined random number generator. RT-5014, IBM Research, Tokyo Research Laboratory, Nov. 1989.

98. Tindo, G. Automates cellulaires; applications à la modélisation de certains systèmes discrets et à la conception d'une architecture parallèle pour la génération de suites pseudo-aléatoires. Thèse de doctorat en informatique, Université de Nantes, France, Jan. 1990.

99. Wichmann, B. A. and Hill, I. D. An Efficient and portable pseudo-random number generator. *Appl. Stat. 31* (1982) 188–190. Also see corrections and remarks in the same journal by Wichmann and Hill *33* 123; (1984) McLeod *34* (1985) 198–200; Zeisel *35* (1986) 89.

100. Wichmann, B. A. and Hill, I. D. Building a random number generator. *Byte 12,* 3 (1987) 127–128.

101. Wikramaratna, R. S. ACORN—A new method for generating sequences of uniformly distributed pseudo-random numbers. *J. Comput. Phys. 83* (1989) 16–31.

CR Categories and Subject Descriptors: G.3 **[Probability and Statistics]:** *Random number generation*

General Terms: Algorithms

Additional Key Words and Phrases: Simulation, uniform variate generation, pseudorandom numbers

About the Author:

PIERRE L'ECUYER is a professor at the Université de Montréal.

Author's Present Address: Département d'informatique et de recherche opérationnelle, Université de Montréal, C.P. 6128, Succ. A, Montréal, Canada, H3C 3J7.

This work has been supported by NSERC-Canada Grant #A5463 and FCAR-Quebec Grant #EQ2831.