CHAPTER 4

RANDOM NUMBER GENERATION[1]

Pierre L'Ecuyer

Université de Montréal[2]

4.1 INTRODUCTION

Random numbers are the nuts and bolts of simulation. Typically, all the randomness required by the model is *simulated* by a random number generator whose output is *assumed* to be a sequence of independent and identically distributed (IID) $U(0,1)$ random variables (i.e., continuous random variables distributed uniformly over the interval $(0,1)$). These *random numbers* are then transformed as needed to simulate random variables from different probability distributions, such as the normal, exponential, Poisson, binomial, geometric, discrete uniform, etc., as well as multivariate distributions and more complicated random objects. In general, the validity of the transformation methods depends strongly on the IID $U(0,1)$ assumption. But this assumption is *false*, since the random number generators are actually simple deterministic programs trying to fool the user by producing a deterministic sequence that *looks* random.

What could be the impact of this on the simulation results? Despite this problem, are there "safe" generators? What about the generators commonly available in system libraries and simulation packages? If they are not satisfactory, how can we build better ones? Which ones should be used, and where is the code? These are some of the topics addressed in this chapter.

---

[1]Chapter 4 of the *Handbook on Simulation*, Ed.: Jerry Banks, Wiley, 1998. Version: February 27, 1998

[2]Département d'informatique et de recherche opérationnelle, Université de Montréal, C.P. 6128, Succ. Centre-Ville, Montréal, H3C 3J7, Canada. `Email: lecuyer@iro.umontreal.ca`

4.1.1 Pseudorandom Numbers

To draw the winning number for several million dollars in a lottery, people would generally not trust a computer. They would rather prefer a simple physical system that they understand well, such as drawing balls from one or more container(s) to select the successive digits of the number (as done, for example, by Loto Quebec each week in Montreal). Even this requires many precautions: The balls must have identical weights and sizes, be well mixed, and be changed regularly to reduce the chances that some numbers come out more frequently than others in the long run. Such a procedure is clearly not practical for computer simulations, which often require millions and millions of random numbers.

Several other physical devices to produce random noise have been proposed and experiments have been conducted using these generators. These devices include gamma ray counters, noise diodes, and so on [47, 62]. Some of these devices have been commercialized and can be purchased to produce random numbers on a computer. But they are cumbersome and they may produce *unsatisfactory* outputs, as there may be significant correlation between the successive numbers. Marsaglia [90] applied a battery of statistical tests to three such commercial devices recently and he reports that all three failed the tests spectacularly.

As of today, the most convenient and most reliable way of generating the random numbers for stochastic simulations appears to be via deterministic algorithms with a solid mathematical basis. These algorithms produce a sequence of numbers which are in fact not random at all, but seem to behave like independent random numbers; that is, like a realization of a sequence of IID $U(0,1)$ random variables. Such a sequence is called *pseudorandom* and the program that produces it is called a *pseudorandom number generator*. In simulation contexts, the term *random* is used instead of *pseudorandom* (a slight abuse of language, for simplification) and we do so in this chapter. The following definition is taken from L'Ecuyer [62, 64].

Definition 1 A *(pseudo)random number generator* is a structure $\mathcal{G} = (S, s_0, T, U, G)$, where S is a finite set of *states*, $s_0 \in S$ is the *initial state* (or *seed*), the mapping

$T : S \to S$ is the *transition function*, U is a finite set of *output* symbols, and $G : S \to U$ is the *output function*.

The state of the generator is initially s_0 and evolves according to the recurrence $s_n = T(s_{n-1})$, for $n = 1, 2, 3, \ldots$. At step n, the generator outputs the number $u_n = G(s_n)$. The u_n, $n \geq 0$, are the *observations*, and are also called the *random numbers* produced by the generator. Clearly, the sequence of states s_n is eventually periodic, since the state space S is finite. Indeed, the generator must eventually revisit a state previously seen; that is, $s_j = s_i$ for some $j > i \geq 0$. From then on, one must have $s_{j+n} = s_{i+n}$ and $u_{j+n} = u_{i+n}$ for all $n \geq 0$. The *period* length is the smallest integer $\rho > 0$ such that for some integer $\tau \geq 0$ and for all $n \geq \tau$, $s_{\rho+n} = s_n$. The smallest τ with this property is called the *transient*. Often, $\tau = 0$ and the sequence is then called *purely periodic*. Note that the period length cannot exceed $|S|$, the cardinality of the state space. Good generators typically have their ρ very close to $|S|$ (otherwise, there is a waste of computer memory).

4.1.2 Example: A Linear Congruential Generator

Example 1 The best-known and (still) most widely used types of generators are the simple linear congruential generators (LCGs) [41, 57, 60, 82]. The state at step n is an integer x_n and the transition function T is defined by the recurrence

$$x_n = (a x_{n-1} + c) \bmod m, \tag{1}$$

where $m > 0$, $a > 0$, and c are integers called the *modulus*, the *multiplier*, and the *additive constant*, respectively. Here, "mod m" denotes the operation of taking the least nonnegative residue modulo m. In other words, multiply x_{n-1} by a, add c, divide the result by m, and put x_n equal to the remainder of the division. One can identify s_n with x_n and the state space S is the set $\{0, \ldots, m-1\}$. To produce values in the interval $[0, 1]$, one can simply define the output function G by $u_n = G(x_n) = x_n/m$.

When $c = 0$, this generator is called a *multiplicative linear congruential generator* (MLCG). The maximal period length for the LCG is m in general. For the MLCG it cannot exceed $m - 1$, since $x_n = 0$ is an absorbing state that must be avoided. Two popular values of m are $m = 2^{31} - 1$ and $m = 2^{32}$. But as discussed later, these values are too small for the requirements of today's simulations. LCGs with such small moduli are still in widespread use, mainly because of their simplicity and ease of implementation, but we believe that they should be discarded and replaced by more robust generators.

For a concrete illustration, let $m = 2^{31} - 1 = 2147483647$, $c = 0$, and $a = 16807$. These parameters were originally proposed in [83]. Take $x_0 = 12345$. Then

$$\begin{aligned}
x_1 &= 16807 \times 12345 \bmod m = 207482415, \\
u_1 &= 207482415/m = 0.0966165285, \\
x_2 &= 16807 \times 207482415 \bmod m = 1790989824, \\
u_2 &= 1790989824/m = 0.8339946274, \\
x_3 &= 16807 \times 1790989824 \bmod m = 2035175616, \\
u_3 &= 2035175616/m = 0.9477024977,
\end{aligned}$$

and so on.

4.1.3 Seasoning the Sequence with External Randomness

In certain circumstances one may want to combine the deterministic sequence with external physical noise. The simplest and most frequently used way of doing this in simulation contexts is to select the seed s_0 randomly. If s_0 is drawn uniformly from S, say by picking balls randomly from a container or by tossing fair coins, the generator can be viewed as an extensor of randomness: It stretches a short, truly random seed into a longer sequence of random-looking numbers. Definition 1 can easily be generalized to accommodate this possibility: Add to the structure a probability distribution μ defined on S and say that s_0 is selected from μ.

In some contexts, one may want to rerandomize the state s_n of the generator every now and then, or to jump ahead from s_n to $s_{n+\nu}$ for some random integer ν. For example, certain types of slot machines in casinos use a simple deterministic random number generator, which keeps running at full speed (i.e., computing its successive states) even when there is nobody playing with the machine. Whenever a player hits the appropriate button and some random numbers are needed to determine the winning combination (e.g., in the game of Keno) or to draw a hand of cards (e.g., for poker machines), the generator provides the output corresponding to its current state. Each time the player hits the button, he or she selects a ν, as just mentioned. This ν is random (although not uniformly distributed). Since typical generators can advance by more than 1 million states per second, hitting the button at the right time to get a specific state or predicting the next output value from the previous ones is almost impossible.

One could go further and select not only the seed, but also some parameters of the generator at random. For example, for a MLCG, one may select the multiplier a at random from a given set of values (for a fixed m) or select the pairs (a, m) at random from a given set. Certain classes of generators for cryptographic applications are defined in a way that the parameters of the recurrence (e.g., the modulus) are viewed as part of the seed and must be generated randomly for the generator to be safe (in the sense of unpredictability).

After observing that physical phenomena by themselves are bad sources of random numbers and that the deterministic generators may produce sequences with too much structure, Marsaglia [90] decided to combine the output of some random number generators with various sources of white and black noise, such as music, pictures, or noise produced by physical devices. The combination was done by addition modulo 2 (bitwise exclusive-or) between the successive bits of the generator's output and of the binary files containing the noise. The result was used to produce a CD-ROM containing 4.8 billion random bits, which appear to behave as independent bits distributed uniformly over the set $\{0, 1\}$. Such a CD-ROM may be interesting but is no universal solution: Its use cannot match the speed and convenience of a good generator, and some applications require much more random numbers than provided on this disk.

4.1.4 Design of Good Generators

How can one build a deterministic generator whose output looks totally random? Perhaps a first idea is to write a computer program more or less at random that can also modify its own code in an unpredictable way. However, experience shows that random number generators should not be built at random (see Knuth [57] for more discussion on this). Building a good random number generator may look easy on the surface, but it is not. It requires a good understanding of heavy mathematics.

The techniques used to evaluate the quality of random number generators can be partitioned into two main classes: The *structural analysis* methods (sometimes called *theoretical tests*) and the *statistical* methods (also called *empirical tests*). An empirical test views the generator as a black box. It observes the output and applies a statistical test of hypothesis to catch up significant statistical defects. An unlimited number of such tests can be designed. Structural analysis, on the other hand, studies the mathematical structure underlying the successsive values produced by the generator, most often over its entire period length. For example, vectors of t successive output values of a LCG can be viewed as points in the t-dimensional unit hypercube $[0,1]^t$. It turns out that all these points, over the entire period of the generator, form a regular *lattice* structure. As a result, all the points lie in a limited number of equidistant parallel hyperplanes, in each dimension t. Computing certain numerical figures of merit for these lattices (e.g., computing the distances between neighboring hyperplanes) is an example of structural analysis. Statistical testing and structural analysis is discussed more extensively in forthcoming sections. We emphasize that all these methods are in a sense heuristic: None ever proves that a particular generator is perfectly random or fully reliable for simulation. The best they can do is improve our confidence in the generator.

4.1.5 Overview of What Follows

We now give an overview of the remainder of this chapter. In the next section we portray our ideal random number generator. The desired properties include uniformity, independence, long period, rapid jump-ahead capability, ease of implementation, and efficiency in terms of speed and space (memory size used). In certain situations, unpredictability is also an issue. We discuss the scope and significance of structural analysis as a guide to select families of generators and choose specific parameters. Section 4.3 covers generators based on linear recurrences. This includes the linear congruential, multiple recursive, multiply-with-carry, Tausworthe, generalized feedback shift register generators, all of which have several variants, and also different types of combinations of these. We study their structural properties at length. Section 4.4 is devoted to methods based on nonlinear recurrences, such as inversive and quadratic congruential generators, as well as other types of methods originating from the field of cryptology. Section 4.5 summarizes the ideas of statistical testing. In Section 4.6 we outline the specifications of a modern uniform random number package and refers to available implementations. We also discuss parallel generators briefly.

4.2 DESIRED PROPERTIES

4.2.1 Unpredictability and "True" Randomness

From the user's perspective, an ideal random number generator should be like a black box producing a sequence that cannot be distinguished from a truly random one. In other words, the goal is that given the output sequence $(u_0, u_1, \ldots)$ and an infinite sequence of IID $U(0,1)$ random variables, no statistical test (or computer program) could tell which is which with probability larger than $1/2$. An equivalent requirement is that after observing any finite number of output values, one cannot guess any given bit of any given unobserved number better than by flipping a fair coin. But this is an impossible dream. The pseudorandom sequence can always be determined by observing it sufficiently, since it is periodic. Similarly, for any periodic sequence, if enough computing time is allowed, it is always possible to construct a statistical test that the sequence will fail spectacularly.

To dilute the goal we may limit the time of observation of the sequence and the computing time for the test. This leads to the introduction of *computational complexity* into the picture. More specifically, we now consider a *family* of generators, $\{\mathcal{G}_k, k = 1, 2, \ldots\}$, indexed by an integral parameter k equal to the number of bits required to represent the state of the generator. We assume that the time required to compute the functions T and G is (at worst) polynomial in k. We also restrict our attention to the class of statistical tests whose running time is polynomial in k. Since the period length typically increases as 2^k, this precludes the tests that exhaust the period. A test is also allowed to toss coins at random, so its outcome is really a random variable. We say that the family $\{\mathcal{G}_k\}$ is *polynomial-time perfect* if, for any polynomial time statistical test trying to distinguish the output sequence of the generator from an infinite sequence of IID $U(0,1)$ random variables, the probability that the test makes the right guess does not exceed $1/2 + e^{-k\epsilon}$, where ϵ is a positive constant. An equivalent requirement is that no polynomial-time algorithm can predict any given bit of u_n with probability of success larger than $1/2 + e^{-k\epsilon}$, after observing $u_0, \ldots, u_{n-1}$, for some $\epsilon > 0$. This setup is based on the idea that what cannot be computed in polynomial time is practically impossible to compute if k is reasonably large. It was introduced in cryptology, where unpredictability is a key issue (see [4, 6, 59, 78] and other references given there).

Are efficient polynomial-time perfect families of generators available? Actually, nobody knows for sure whether or not such a family *exists*. But some generator families are *conjectured* to be polynomial-time perfect. The one with apparently the best behavior so far is the BBS, introduced by Blum, Blum, and Shub [4], explained in the next example.

Example 2 The BBS generator of size k is defined as follows. The state space S_k is the set of triplets (p, q, x) such that p and q are $(k/2)$-bit prime integers, $p+1$ and $q+1$ are both divisible by 4, and x is a quadratic residue modulo $m = pq$, relatively prime to m (i.e., x can be expressed as $x = y^2 \bmod m$ for some integer y that is not divisible by p or q). The initial state (seed) is chosen randomly from S_k, with the uniform distribution. The state then evolves as follows: p and q remain unchanged and the successive values of x follow the recurrence

$$x_n = x_{n-1}^2 \bmod m.$$

At each step, the generator outputs the ν_k least significant bits of x_n (i.e., $u_n = x_n \bmod 2^{\nu_k}$), where $\nu_k \le K \log k$ for some constant K. The relevant conjecture here is that with probability at least $1 - e^{-k\epsilon}$ for some $\epsilon > 0$, factoring m (i.e., finding p or q, given m) cannot be done in polynomial time (in k). Under this conjecture, the BBS generator has been proved polynomial-time perfect [4, 124]. Now, a down-to-earth question is: How large should be k to be safe in practice? Also, how small should be K? Perhaps no one really knows. A k larger than a few thousands is probably pretty safe but makes the generator too slow for general simulation use.

Most of the generators discussed in the remainder of this chapter are known not to be polynomial-time perfect. However, they seem to have good enough statistical properties for most reasonable simulation applications.

4.2.2 What Is a Random Sequence?

The idea of a truly random sequence makes sense only in the (abstract) framework of probability theory. Several authors (see, e.g., [57]) give definitions of a random sequence, but these definitions require nonperiodic infinite-length sequences. Whenever one selects a generator with a fixed seed, as in Definition 1, one always obtains a deterministic sequence of finite length (the length of the period) which repeats itself indefinitely. Choosing such a random number generator then amounts to selecting a finite-length sequence. But among all sequences of length ρ of symbols from the set U, for given ρ and finite U, which ones are better than others? Let $|U|$ be the cardinality of the set U. If all the symbols are chosen uniformly and independently from U, each of the $|U|^\rho$ possible sequences of symbols from U has the same probability of occurring, namely $|U|^{-\rho}$. So it appears that no particular sequence (i.e., no generator) is better than any other. A pretty disconcerting conclusion! To get out of this dead end, one must take a different point of view.

Suppose that a starting index n is randomly selected, uniformly from the set $\{1, 2, \ldots, \rho\}$, and consider the output vector (or subsequence) $\boldsymbol{u}_n = (u_n, \ldots, u_{n+t-1})$, where $t \ll \rho$. Now, $\boldsymbol{u}_n$ is a (truly) random vector. We would like $\boldsymbol{u}_n$ to be uniformly distributed (or almost) over the set U^t of all vectors of length t. This requires $\rho \geq |U|^t$, since there are at most ρ different values of $\boldsymbol{u}_n$ in the sequence. For $\rho < |U|^t$, the set $\Psi = \{\boldsymbol{u}_n, 1 \leq$

$n \leq \rho\}$ can cover only part of the set U^t. Then one may ask Ψ to be uniformly spread over U^t. For example, if U is a discretization of the unit interval $[0,1]$, such as $U = \{0, 1/m, 2/m, \ldots, (m-1)/m\}$ for some large integer m, and if the points of Ψ are evenly distributed over U^t, they are also (pretty much) evenly distributed over the unit hypercube $[0,1]^t$.

Example 3 Suppose that $U = \{0, 1/100, 2/100, \ldots, 99/100\}$ and that the period of the generator is $\rho = 10^4$. Here we have $|U| = 100$ and $\rho = |U|^2$. In dimension 2, the pairs $\boldsymbol{u}_n = (u_n, u_{n+1})$ can be uniformly distributed over U^2, and this happens if and only if each pair of successive values of the form $(i/100,\ j/100)$, for $0 \leq i, j < 100$ occurs exactly once over the period. In dimension $t > 2$, we have $|U|^t = 10^{2t}$ points to cover but can cover only 10^4 of those because of the limited period length of our generator. In dimension 3, for instance, we can cover only 10^4 points out of 10^6. We would like those 10^4 points that are covered to be very uniformly distributed over the unit cube $[0,1]^3$.

An even distribution of Ψ over U^t, in all dimensions t, will be our basis for discriminating among generators. The rationale is that under these requirements, subsequences of any t successive output values produced by the generator, from a random seed, should behave much like random points in the unit hypercube. This captures both *uniformity* and *independence*: If $\boldsymbol{u}_n = (u_n, \ldots, u_{n+t-1})$ is generated according to the uniform distribution over $[0,1]^t$, the components of $\boldsymbol{u}_n$ are independent and uniformly distributed over $[0,1]$. This idea of looking at what happens when the seed is random, for a given finite sequence, is very similar to the *scanning ensemble* idea of Compagner [11, 12], except that we use the framework of probability theory instead.

The reader may have already noticed that under these requirements, Ψ will not look at all like a random set of points, because its distribution over U^t is too even (or *superuniform*, as some authors say [116]). But what the foregoing model assumes is that only a few points are selected at random from the set Ψ. In this case, the best one can do for these points to be distributed approximately as IID uniforms is to take Ψ superuniformly distributed over U^t. For this to make some sense, ρ must be several orders of magnitude larger than the number of output values actually used by the simulation.

To assess this even distribution of the points over the entire period, some (theoretical) understanding of their structural properties is necessary. Generators whose structural properties are well understood and precisely described may look less random, but those that are more complicated and less understood are not necessarily better. They may hide strong correlations or other important defects. One should avoid generators without convincing theoretical support. As a basic requirement, the period length must be known and huge. But this is not enough. Analyzing the equidistribution of the points as just discussed, which is sometimes achieved by studying the lattice structure, usually gives good insight on how the generator behaves. Empirical tests can be applied thereafter, just to improve one's confidence.

4.2.3 Discrepancy

A well-established class of measures of uniformity for finite sequences of numbers are based on the notion of *discrepancy*. This notion and most related results are well covered by Niederreiter [102]. We only recall the most basic ideas here.

Consider the N points $\boldsymbol{u}_n = (u_n, \ldots, u_{n+t-1})$, for $n = 0, \ldots, N-1$, in dimension t, formed by (overlapping) vectors of t successive output values of the generator. For any hyper-rectangular box aligned with the axes, of the form $R = \prod_{j=1}^{t} [\alpha_j,\ \beta_j)$, with $0 \le \alpha_j < \beta_j \le 1$, let $I(R)$ be the number of points $\boldsymbol{u}_n$ falling into R, and $V(R) = \prod_{j=1}^{t} (\beta_j - \alpha_j)$ be the volume of R. Let $\mathcal{R}$ be the set of all such regions R, and

$$D_N^{(t)} = \max_{R \in \mathcal{R}} |V(R) - I(R)/N|.$$

This quantity is called the t-dimensional *(extreme) discrepancy* of the set of points $\{\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{N-1}\}$. If we impose $\alpha_j = 0$ for all j; that is, we restrict $\mathcal{R}$ to those boxes which have one corner at the origin, then the corresponding quantity is called the *star discrepancy*, denoted by $D_N^{*(t)}$. Other variants also exist, with richer $\mathcal{R}$.

A low discrepancy value means that the points are very evenly distributed in the unit hypercube. To get superuniformity of the sequence over its entire period, one might want to *minimize* the discrepancy $D_\rho^{(t)}$ or $D_\rho^{*(t)}$ for $t = 1, 2, \ldots$. A major practical difficulty with discrepancy is that it can be computed only for very special cases. For LCGs, for example, it can be computed efficiently in dimension $t = 2$, but for larger t, the computing cost then increases as $O(N^t)$. In most cases, only (upper and lower) bounds on the discrepancy are available. Often, these bounds are expressed as orders of magnitude as a function of N, are defined for $N = \rho$, and/or are averages over a large (specific) class of generators (e.g., over all full-period MLCGs with a given prime modulus). Discrepancy also depends on the rectangular orientation of the axes, in contrast to other measures of uniformity, such as the distances between hyperplanes for LCGs (see Section 4.3.4). On the other hand, it applies to all types of generators, not only those based on linear recurrences.

We previously argued for superuniformity over the entire period, which means seeking the lowest possible discrepancy. When a subsequence of length N is used (for $N \ll \rho$), starting, say, at a random point along the entire sequence, the discrepancy of that subsequence should behave (viewed as a random variable) as the discrepancy of a sequence of IID $U(0,1)$ random variables. The latter is (roughly) of order $O(N^{-1/2})$ for both the star and extreme discrepancies.

Niederreiter [102] shows that the discrepancy of full-period MLCGs *over their entire period* (of length $\rho = m - 1$), on the average over multipliers a, is of order $O(m^{-1}(\log m)^t \log\log(m+1))$. This order is much smaller (for large m) than $O(m^{-1/2})$, meaning superuniformity. Over small fractions of the period length, the available bounds on the discrepancy are more in accordance with the law of the iterated logarithm [100]. This is yet another important justification for never using more than a negligible fraction of the period.

Suppose now that numbers are generated in $[0,1]$ with L fractional binary digits. This gives *resolution* 2^{-L}, which means that all u_n's are multiples of 2^{-L}. It then follows ([102]) that $D_N^{*(t)} \geq 2^{-L}$ for all $t \geq 1$ and $N \geq 1$. Therefore, as a *necessary* condition for the discrepancy to be of the right order of magnitude, the resolution 2^{-L} must be small enough for the number of points N that we plan to generate: 2^{-L} should be much smaller than $N^{-1/2}$. A too coarse discretization implies a too large discrepancy.

4.2.4 Quasi-random Sequences

The interest in discrepancy stems largely from the fact that deterministic error bounds for (Monte Carlo) numerical integration of a function are available in terms of $D_N^{(t)}$ and of a certain measure of variability of the function. In that context, the smaller the discrepancy, the better (because the aim is to minimize the numerical error, not really to imitate IID $U(0,1)$ random variables). Sequences for which the discrepancy of the first N values is small for all N are called *low-discrepancy* or *quasi-random* sequences [102]. Numerical integration using such sequences is called *quasi-Monte Carlo integration*. To estimate the integral using N points, one simply evaluates the function (say, a function of t variables) at the first N points of the sequence, takes the average, multiplies by the volume of the domain of integration, and uses the result as an approximation of the integral. Specific low-discrepancy sequences have been constructed by Sobol', Faure, and Niederreiter, among others (see [102]). Owen [106] gives a recent survey of their use. In this chapter we concentrate on *pseudorandom* sequences and will not discuss *quasi-random* sequences further.

4.2.5 Long Period

Let us now return to the desired properties of pseudorandom sequences, starting with the length of the period. What is long enough? Suppose that a simulation experiment takes N random numbers from a sequence of length ρ. Several reasons justify the need to take $\rho \gg N$ (see, e.g., [21, 64, 86, 102, 112]). Based on geometric arguments, Ripley [112] suggests that $\rho \gg N^2$ for linear congruential generators. The papers [75, 79] provide strong experimental support for this, based on extensive empirical tests. Our previous discussion also supports the view that ρ must be huge in general.

Period lengths of 2^{32} or smaller, which are typical for the default generators of many operating systems and software packages, are unacceptably too small. Such period lengths can be exhausted in a matter of minutes on today's workstations. Even $\rho = 2^{64}$ is a relatively small period length. Generators with period lengths over 2^{200} are now available.

4.2.6 Efficiency

Some say that the speed of a random number generator (the number of values that it can generate per second, say) is not very important for simulation, since generating the numbers typically takes only a tiny fraction of the simulation time. But there are several counterexamples, such as for certain large simulations in particle physics [26], or when using intensive Monte Carlo simulation to estimate with precision the distribution of a statistic that is fast to compute but requires many random numbers. Moreover, even if a fast generator takes only, say, 5% of the simulation time, changing to another one that is 20 times slower will approximately double the total simulation time. Since simulations often consume several hours of CPU time, this is significant.

The memory size used by a generator might also be important in general, especially since simulations often use several generators in parallel, for instance to maintain synchronization for variance reduction purposes (see Section 4.6 and [7, 60] for more details).

4.2.7 Repeatability, Splitting Facilities, and Ease of Implementation

The ability to replicate exactly the same sequence of random numbers, called *repeatability*, is important for program verification and to facilitate the implementation of certain variance reduction techniques [7, 55, 60, 113]. Repeatability is a major advantage of pseudorandom sequences over sequences generated by physical devices. The latter can of course be stored on disks or other memory devices, and then reread as needed, but this is less convenient than a good pseudorandom number generator that fits in a few lines of code in a high-level language.

A code is said to be *portable* if it works without change and produces exactly the same sequence (at least up to machine accuracy) across all "standard" compilers and computers. A portable code in a high-level language is clearly much more convenient than a machine-dependent assembly-language implementation, for which repeatability is likely to be more difficult to achieve.

Ease of implementation also means the ease of *splitting* the sequence into (long) disjoint substreams and jumping quickly from one substream to the next. In Section 4.6 we show why this is important. For this, there should be an efficient way to compute the state $s_{n+\nu}$ for any large ν, given s_n. For most linear-type generators, we know how to do that. But for certain types of nonlinear generators and for some methods of combination (such as *shuffling*), good jump-ahead techniques are unknown. Implementing a random number package as described in Section 4.6 requires efficient jump-ahead techniques.

4.2.8 Historical Accounts

There is an enormous amount of scientific literature on random number generation. Law and Kelton [60] present a short (but interesting) historical overview. Further surveys and historical accounts of the old days are provided in [47, 53, 119].

Early attempts to construct pseudorandom number generators have given rise to all sorts of bad designs, sometimes leading to disatrous results. An illustrative example is the *middle-square* method, which works as follows (see, e.g., [57, 60]). Take a b-digit number x_{i-1} (say, in base 10, with b even), square it to obtain a $2b$-digit number (perhaps with zeros on the left), and extract the b middle digits to define the next number x_i. To obtain an output value u_i in $[0, 1)$, divide x_i by 10^b. The period length of this generator depends on the initial value and is typically very short, sometimes of length 1 (such as when the sequence reaches the absorbing state $x_i = 0$). Hopefully, it is no longer used. Another example of a bad generator is RANDU (see G4 in Table 1).

4.3 LINEAR METHODS

4.3.1 Multiple-Recursive Generator

Consider the linear recurrence

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod m, \tag{2}$$

where the *order* k and the *modulus* m are positive integers, while the *coefficients* $a_1, \ldots, a_k$ are integers in the range $\{-(m-1), \ldots, m-1\}$. Define $\mathbb{Z}_m$ as the set $\{0, 1, \ldots, m-1\}$ on which operations are performed modulo m. The state at step n of the *multiple recursive generator* (MRG) [57, 62, 102] is the vector $s_n = (x_n, \ldots, x_{n+k-1}) \in \mathbb{Z}_m^k$. The output function can be defined simply by $u_n = G(s_n) = x_n/m$, which gives a value in $[0, 1]$, or by a more refined transformation if a better resolution than $1/m$ is required. The special case where $k = 1$ is the MLCG mentioned previously.

The characteristic polynomial P of (2) is defined by

$$P(z) = z^k - a_1 z^{k-1} - \cdots - a_k. \tag{3}$$

The maximal period length of (2) is $\rho = m^k - 1$, reached if and only if m is prime and P is a primitive polynomial over $\mathbb{Z}_m$, identified here as the finite field with m elements. Suppose that m is prime and let $r = (m^k - 1)/(m - 1)$. The polynomial P is primitive over $\mathbb{Z}_m$ if and only if it satisfies the following conditions, where everything is assumed to be modulo m (see [57])

(a) $[(-1)^{k+1} a_k]^{(m-1)/q} \neq 1$ for each prime factor q of $m - 1$

(b) $z^r \bmod P(z) = (-1)^{k+1} a_k$

(c) $z^{r/q} \bmod P(z)$ has degree > 0 for each prime factor q of r, $1 < q < r$.

For $k = 1$ and $a = a_1$ (the MLCG case), these conditions simplify to $a \neq 0 \pmod m$ and $a^{(m-1)/q} \neq 1 \pmod m$ for each prime factor q of $m-1$. For large r, finding the factors q to check condition (c) can be too difficult, since it requires the factorization of r. In this case, the trick is to choose m and k so that r is prime (this can be done only for prime k). Testing primality of large numbers (using probabilistic algorithms, for example, as in [73, 111]) is much easier than factoring. Given m, k, and the factorizations of $m-1$ and r, primitive polynomials are generally easy to find, simply by random search.

If m is not prime, the period length of (2) has an upper bound typically much lower than $m^k - 1$. For $k = 1$ and $m = 2^e$, $e \geq 4$, the maximum period length is 2^{e-2}, which is reached if $a_1 = 3$ or $5 \pmod 8$ and x_0 is odd [57, p. 20]. Otherwise, if $m = p^e$ for p prime and $e \geq 1$, and $k > 1$ or $p > 2$, the upper bound is $(p^k - 1)p^{e-1}$ [36]. Clearly, $p = 2$ is very convenient from the implementation point of view, because the modulo operation then amounts to chopping-off the higher-order bits. So to compute $ax \bmod m$ in that case, for example with $e = 32$ on a 32-bit computer, just make sure that the overflow-checking option or the compiler is turned off, and compute the product ax using unsigned integers while ignoring the overflow.

However, taking $m = 2^e$ imposes a big sacrifice on the period length, especially for $k > 1$. For example, if $k = 7$ and $m = 2^{31} - 1$ (a prime), the maximal period length is $(2^{31}-1)^7 - 1 \approx 2^{217}$. But for $m = 2^{31}$ and the same value of k, the upper bound becomes $\rho \leq (2^7 - 1)2^{31-1} < 2^{37}$, which is more than 2^{180} times shorter. For $k = 1$ and $p = 2$, an upper bound on the period length of the ith least significant bit of x_n is $\max(1, 2^{i-2})$ [7], and if a full cycle is split into 2^d equal segments, all segments are identical except for their d most significant bits [20, 26]. For $k > 1$ and $p = 2$, the upper bound on the period length of the ith least significant bit is $(2^k - 1)2^{i-1}$. So the low-order bits are typically much too regular when $p = 2$. For $k = 7$ and $m = 2^{31}$, for example, the least significant bit has period length at most $2^7 - 1 = 127$, the second least significant bit has period length at most $2(2^7 - 1) = 254$, and so on.

Example 4 Consider the recurrence $x_n = 10205 x_{n-1} \bmod 2^{15}$, with $x_0 = 12345$. The first eight values of x_n, in base 10 and in base 2, are

$$x_0 = 12345 = 011000000111001_2$$
$$x_1 = 20533 = 101000000110101_2$$

$$
\begin{aligned}
x_2 &= 20673 &= 101000011000001_2 \\
x_3 &= 7581 &= 001110110011101_2 \\
x_4 &= 31625 &= 111101110001001_2 \\
x_5 &= 1093 &= 000010001000101_2 \\
x_6 &= 12945 &= 011001010010001_2 \\
x_7 &= 15917 &= 011111000101101_2.
\end{aligned}
$$

The last two bits are always the same. The third least significant bit has a period length of 2, the fourth least significant bit has a period length of 4, and so on.

Adding a constant c as in (1) can slightly increase the period-length. The LCG with recurrence (1) has period length m if and only if the following conditions are satisfied ([57, p. 16])

1. c is relatively prime to m.
2. $a - 1$ is a multiple of p for every prime factor p of m (including m itself if m is prime).
3. If m is a multiple of 4, then $a - 1$ is also a multiple of 4.

For $m = 2^e \geq 4$, these conditions simplify to c is odd and $a \bmod 4 = 1$. But the low-order bits are again too regular: The period length of the ith least significant bit of x_n is at most 2^i.

A constant c can also be added to the right side of the recurrence (2). One can show (see [62]) that a linear recurrence of order k with such a constant term is equivalent to some linear recurrence of order $k+1$ with no constant term. As a result, an upper bound on the period length of such a recurrence with $m = p^e$ is $(p^{k+1} - 1)p^{e-1}$, which is much smaller than m^k for large e and k.

All of this argues against the use of power-of-2 moduli in general, despite their advantage in terms of implementation. It favors prime moduli instead. Later, when discussing combined generators, we will also be interested in moduli that are the products of a few large primes.

4.3.2 Implementation for Prime m

For $k > 1$ and prime m, for the characteristic polynomial P to be primitive, it is necessary that a_k and at least another coefficient a_j be nonzero. From the implementation point of view, it is best to have only two nonzero coefficients; that is, a recurrence of the form

$$x_n = (a_r x_{n-r} + a_k x_{n-k}) \bmod m \tag{4}$$

with characteristic trinomial P defined by $P(z) = z^k - a_r z^{k-r} - a_k$. Note that replacing r by $k - r$ generates the same sequence in reverse order.

When m is not a power of 2, computing and adding the products modulo m in (2) or (4) is not necessarily straightforward, using ordinary integer arithmetic, because of the possibility of overflow: The products can exceed the largest integer representable on the computer. For example, if $m = 2^{31} - 1$ and $a_1 = 16807$, then x_{n-1} can be as large as $2^{31} - 2$, so the product $a_1 x_{n-1}$ can easily exceed 2^{31}. L'Ecuyer and Côté [76] study and compare different techniques for computing a product modulo a large integer m, using only integer arithmetic, so that no intermediate result ever exceeds m. Among the *general* methods, working for all representable integers and easily implementable in a high-level language, *decomposition* was the fastest in their experiments. Roughly, this method simply decomposes each of the two integers that are to be multiplied in two blocks of bits (e.g., the 15 least significant bits and the 16 most significant ones, for a 31-bit integer) and then cross-multiplies the blocks and adds (modulo m) just as one does when multiplying large numbers by hand.

There is a faster way to compute $ax \bmod m$ for $0 < a, x < m$, called *approximate factoring*, which works under the condition that

$$a\,(m \bmod a) < m. \tag{5}$$

This condition is satisfied if and only if $a = i$ or $a = \lfloor m/i \rfloor$ for $i < \sqrt{m}$ (here $\lfloor x \rfloor$ denotes the largest integer smaller or equal to x, so $\lfloor m/i \rfloor$ is the integer division of m by i). To implement the approximate factoring method, one initially precomputes (once for all) the constants $q = \lfloor m/a \rfloor$ and $r = m \bmod a$. Then, for any positive integer $x < m$, the following instructions have the same effect as the assignment $x \leftarrow ax \bmod m$, but with all intermediate (integer) results remaining strictly between $-m$ and m [7, 61, 107]:

$$
\begin{aligned}
&y \leftarrow \lfloor x/q \rfloor; \\
&x \leftarrow a(x - yq) - yr; \\
&\text{IF } x < 0 \text{ THEN } x \leftarrow x + m \text{ END.}
\end{aligned}
$$

As an illustration, if $m = 2^{31} - 1$ and $a = 16807$, the generator satisfies the condition, since $16807 < \sqrt{m}$. In this case, one has $q = 127773$ and $r = 2836$.

Hörmann and Derflinger [51] give a different method, which is about as fast, for the case where $m = 2^{31} - 1$. Fishman [41, p. 604] also uses a different method to implement the LCG with $m = 2^{31} - 1$ and $a = 95070637$, which does not satisfy (5).

Another approach is to represent all the numbers and perform all the arithmetic modulo m in double-precision floating point. This works provided that the multipliers a_i are small enough so that the integers $a_i x_{n-i}$ and their sum are always represented *exactly* by the floating-point values. A sufficient condition is that the floating-point numbers are represented with at least

$$\lceil \log_2 ((m-1)(a_1 + \cdots + a_k)) \rceil$$

bits of precision in their mantissa, where $\lceil x \rceil$ denotes the smallest integer larger or equal to x. On computers with good 64-bit floating-point hardware (most computers nowadays), this approach usually gives by far the fastest implementation (see, e.g., [68] for examples and timings).

4.3.3 Jumping Ahead

To jump ahead from x_n to $x_{n+\nu}$ with an MLCG, just use the relation

$$x_{n+\nu} = a^\nu x_n \bmod m = (a^\nu \bmod m) x_n \bmod m.$$

If many jumps are to be performed with the same ν, the constant $a^\nu \bmod m$ can be precomputed once and used for all subsequent computations.

Example 5 Again, let $m = 2147483647$, $a = 16807$, and $x_0 = 12345$. Suppose that we want to compute x_3 directly from x_0, so $\nu = 3$. One easily finds that $16807^3 \bmod m = 1622650073$ and $x_3 = 1622650073 x_0 \bmod m = 2035175616$, which agrees with the value given in Example 1. Of course, we are usually interested in much larger values of ν, but the method works the same way.

For the LCG, with $c \neq 0$, one has

$$x_{n+\nu} = \left(a^\nu x_n + \frac{c(a^\nu - 1)}{a - 1} \right) \bmod m.$$

To jump ahead with the MRG, one way is to use the fact that it can be represented as a matrix MLCG: $X_n = AX_{n-1} \bmod m$, where X_n is s_n represented as a column vector and A is a $k \times k$ square matrix. Jumping ahead is then achieved in the same way as for the MLCG:

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

Another way is to transform the MRG into its polynomial representation [64], in which jumping ahead is easier, and then apply the inverse transformation to recover the original representation.

4.3.4 Lattice Structure of LCGs and MRGs

A lattice of dimension t, in the t-dimensional real space $\mathbb{R}^t$, is a set of the form

$$L = \left\{ V = \sum_{j=1}^{t} z_j V_j \mid \text{ each } z_j \in \mathbb{Z} \right\}, \tag{6}$$

where $\mathbb{Z}$ is the set of all integers and $\{V_1, \ldots, V_t\}$ is a basis of $\mathbb{R}^t$. The lattice L is thus the set of all *integer* linear combinations of the vectors $V_1, \ldots, V_t$, and these vectors are called a *lattice basis* of L. The basis $\{W_1, \ldots, W_t\}$ of $\mathbb{R}^t$ which satisfies $V_i' W_j = \delta_{ij}$ for all $1 \leq i, j \leq t$ (where the prime means "transpose" and where $\delta_{ij} = 1$ if $i = j$, 0 otherwise) is called the *dual* of the basis $\{V_1, \ldots, V_t\}$, and the lattice generated by this dual basis is called the dual lattice to L.

Consider the set

$$T_t = \{\boldsymbol{u}_n = (u_n, \ldots, u_{n+t-1}) \mid n \geq 0,\ s_0 = (x_0, \ldots, x_{k-1}) \in \mathbb{Z}_m^k\} \tag{7}$$

of all overlapping t-tuples of successive values produced by (2), with $u_n = x_n/m$, from all possible initial seeds. Then this set T_t is the intersection of a lattice L_t with the t-dimensional unit hypercube $I^t = [0,1)^t$. For more detailed studies and to see how to construct a basis for this lattice L_t and its dual, see [23, 57, 73, 77]. For $t \leq k$ it is clear from the definition of T_t that each vector $(x_0, \ldots, x_{t-1})$ in $\mathbb{Z}_m^t$ can be taken as s_0, so $T_t = \mathbb{Z}_m^t/m = (\mathbb{Z}^t/m) \cap I^t$; that is, L_t is the set of all t-dimensional vectors whose coordinates are multiples of $1/m$, and T_t is the set of m^t points in L_t whose coordinates belong to $\{0, 1/m, \ldots, (m-1)/m\}$. For a full-period MRG, this also holds if we fix s_0 in the definition of T_t to any nonzero vector of $\mathbb{Z}_m^k$, and then add the zero vector to T_t. In dimension $t > k$, the set T_t contains only m^k points, while $\mathbb{Z}_m^t/m$ contains m^t points. Therefore, for large t, T_t contains only a small fraction of the t-dimensional vectors whose coordinates are multiples of $1/m$.

For full-period MRGs, the generator covers all of T_t except the zero state in one cycle. In other cases, such as for MRGs with nonprime moduli or MLCGs with power-of-2 moduli, each cycle covers only a smaller subset of T_t, and the lattice generated by that subset is often equal to L_t, but may in some cases be a strict *sublattice* or *subgrid* (i.e., a *shifted lattice* of the form $V_0 + L$ where $V_0 \in \mathbb{R}^t$ and L is a lattice). In the latter case, to analyze the structural properties of the generator, one should examine the appropriate sublattice or subgrid instead of L_t. Consider, for example, an MLCG for which m is a power of 2, $a \bmod 8 = 5$, and x_0 is odd. The t-dimensional points constructed from successive values produced by this generator form a subgrid of L_t containing one-fourth of the points [3, 50]. For a LCG with m a power of 2 and $c \neq 0$, with full period length $\rho = m$, the points all lie in a grid that is a shift of the lattice L_t associated with the corresponding MLCG (with the same a amd m). The value of c determines only the shifting and has no other effect on the lattice structure.

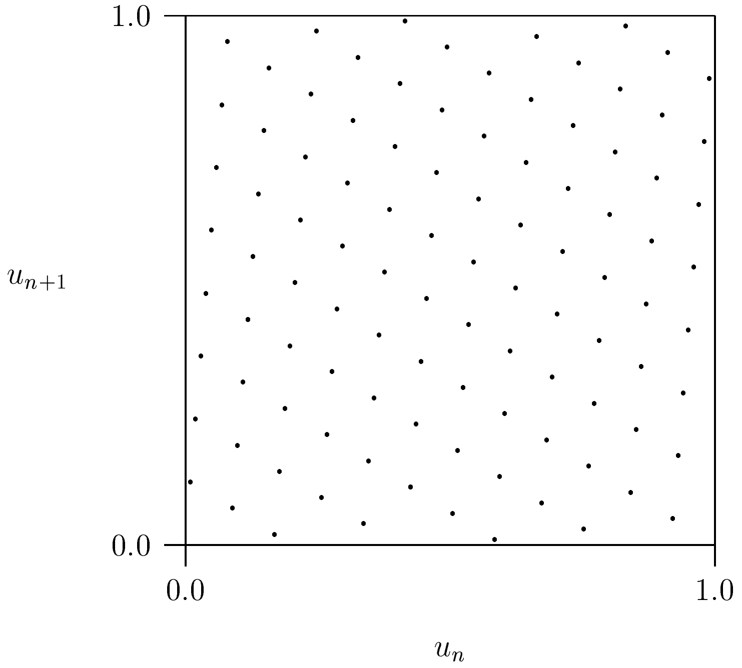

Figure 1: All pairs (u_n, u_{n+1}) for the LCG with $m = 101$ and $a = 12$.

Example 6 Figures 1 to 3 illustrate the lattice structure of a small, but instructional, LCGs with (prime) modulus $m = 101$ and full period length $\rho = 100$, in dimension $t = 2$. They show all 100 pairs of successive values (u_n, u_{n+1}) produced by these generators, for the multipliers $a = 12$, $a = 7$, and $a = 51$, respectively. In each case, one clearly sees the lattice structure of the points. Any pair of vectors forming a basis determine a parallelogram of area $1/101$. This holds more generally: In dimension t, the vectors of any basis of L_t determine a parallelepiped of volume $1/m^k$. Conversely, any set of t vectors that determine such a parallelepiped form a lattice basis.

The points are much more evenly distributed in the square for $a = 12$ than for $a = 51$, and slightly more evenly distributed for $a = 12$ than for $a = 7$. The points of L_t are generally more evenly distributed when there exists a basis comprised of vectors of similar lengths. One also sees from the figures that all the points lie in a relative small number of equidistant parallel lines. In Figure 3, only two lines contain all the points and this leaves large empty spaces between the lines, which is bad.

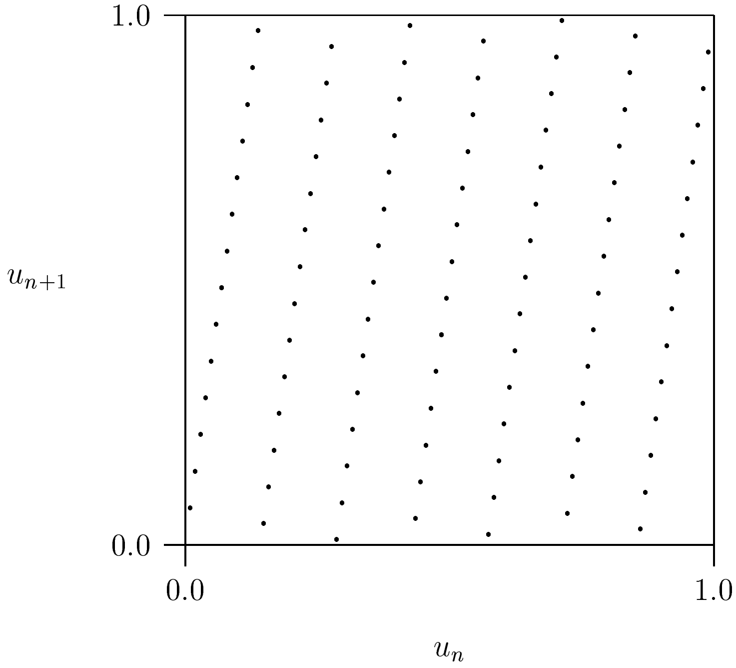

Figure 2: All pairs (u_n, u_{n+1}) for the LCG with $m = 101$ and $a = 7$.

In general, the lattice structure implies that all the points of T_t lie on a family of equidistant parallel hyperplanes. Among all such families of parallel hyperplanes that cover all the points, take the one for which the successive hyperplanes are farthest apart. The distance d_t between these successive hyperplanes is equal to $1/\ell_t$, where ℓ_t is the length of a shortest nonzero vector in the *dual* lattice to L_t. Computing a shortest nonzero vector in a lattice L means finding the combination of values of z_j in (6) giving the shortest V. This is a quadratic optimization problem with integer variables and can be solved by a branch-and-bound algorithm, as in [15, 40]. In these papers the authors use an ellipsoid method to compute the bounds on the z_j for the branch-and-bound. This appears to be the best (general) approach known to date and is certainly much faster than the algorithm given in [23] and [57]. This idea of analyzing d_t was introduced by Coveyou and MacPherson [18] through the viewpoint of spectral analysis. For this historical reason, computing d_t is often called the *spectral test*.

The shorter the distance d_t, the better, because a large d_t means thick empty slices

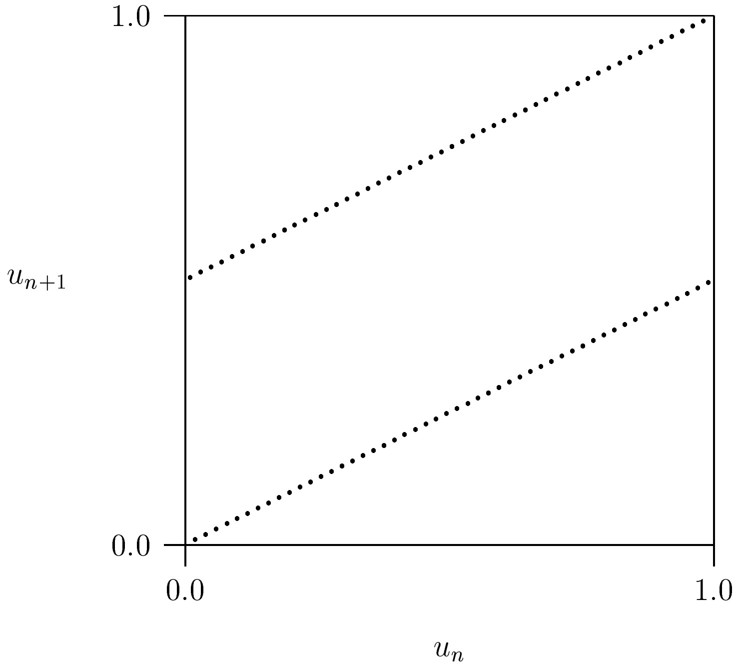

Figure 3: All pairs (u_n, u_{n+1}) for the LCG with $m = 101$ and $a = 51$.

of space between the hyperplanes. One has the theoretical lower bound

$$d_t \geq d_t^* = \frac{1}{\gamma_t m^{k/t}}, \tag{8}$$

where γ_t is a constant which depends only on t and whose exact value is currently known only for $t \leq 8$ [57]. So, for $t \leq 8$ and $T \leq 8$, 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. Values close to 1 are desired. Another lower bound on d_t, for $t > k$, is (see [67])

$$d_t \geq \left(1 + \sum_{j=1}^{k} a_j^2\right)^{-1/2}. \tag{9}$$

This means that an MRG whose coefficients a_j are small is guaranteed to have a large (bad) d_t.

Other figures of merit have been introduced to measure the quality of random number generators in terms of their lattice structure. For example, one can count the minimal number of hyperplanes that contain all the points or compute the ratio of lengths of the shortest and longest vectors in a Minkowski-reduced basis of the lattice. For more details on the latter, which is typically much more costly to compute than d_t, the reader can consult [77] and the references given there. These alternative figures of merit do not tell us much important information in addition to d_t.

Tables 1 and 2 give the values of d_t and S_t for certain LCGs and MRGs. All these generators have full period length. The LCGs of the first table are well known and most are (or have been) heavily used. For $m = 2^{31} - 1$, the multiplier $a = 742938285$ was found by Fishman and Moore [42] in an exhaustive search for the MLCGs with the best value of M_6 for this value of m. It is used in the GPSS/H simulation environment. The second multiplier, $a = 16807$, was originally proposed in [83], is suggested in many simulation books and papers (e.g., [7, 107, 114]) and appears in several software systems such as the SLAM II and SIMAN simulation programming languages, MATLAB [94], the IMSL statistical library [54], and in operating systems for the IBM and Macintosh computers. It satisfies condition (5). The IMSL library also has available the two multipliers 397204094 and 950706376, with the same modulus, as well as the possibility of adding a shuffle to the LCG. The multiplier $a = 630360016$ was proposed in [108], is recommended in [60, 92] among others, and is used in software such as the SIMSCRIPT II.5 and INSIGHT simulation programming languages. Generator G4, with modulus $m = 2^{31}$ and multiplier $a = 65539$, is the infamous RANDU generator, used for a long time in the IBM/360 operating system. Its lattice structure is particularly bad in dimension 3, where all the points lie in only 15 parallel planes. Law and Kelton [60] give a graphical illustration. Generator G5, with $m = 2^{32}$, $a = 69069$, and $c = 1$, is used in the VAX/VMS operating system. The LCG G6, with modulus $m = 2^{48}$, multiplier $a = 25214903917$, and constant $c = 11$, is the generator implemented in the procedure `drand48` of the SUN Unix system's library [117]. G7, whose period length is slighly less than 2^{40}, is used in the *Maple* mathematical software. We actually recommend *none* of the generators G1 to G7. Their period lengths are too short and they fail many statistical tests (see Section 4.5).

In Table 2, G8 and G9 are two MRGs of order 7 found by a random search for multipliers with a "good" lattice structure in all dimensions $t \leq 20$, among those giving

Table 1: Distances between hyperplanes for some LCGs

	G1	G2	G3	G4	G5	G6	G7
m	$2^{31}-1$	$2^{31}-1$	$2^{31}-1$	2^{31}	2^{32}	2^{48}	$10^{12}-11$
k	1	1	1	1	1	1	1
a	742938285	16807	630360016	65539	69069	25214903917	427419669081
c	0	0	0	0	1	11	0
ρ	$2^{31}-2$	$2^{31}-2$	$2^{31}-2$	2^{29}	2^{32}	2^{48}	$10^{12}-12$
S_2	0.8673	0.3375	0.8212	0.9307	0.6541	0.5110	0.7513
S_3	0.8607	0.4412	0.4317	0.0119	0.4971	0.8030	0.7366
S_4	0.8627	0.5752	0.7833	0.0595	0.6223	0.4493	0.6491
S_5	0.8319	0.7361	0.8021	0.1570	0.6583	0.5847	0.7307
S_6	0.8341	0.6454	0.5700	0.2927	0.3356	0.6607	0.6312
S_7	0.6239	0.5711	0.6761	0.4530	0.4499	0.8025	0.5598
S_8	0.7067	0.6096	0.7213	0.6173	0.6284	0.5999	0.5558
$1/m$	4.65E-10	4.65E-10	4.65E-10	4.65E-10	2.33E-10	3.55E-15	1.00E-12
d_2	2.315E-5	5.950E-5	2.445E-5	4.315E-5	3.070E-5	1.085E-7	1.239E-6
d_3	8.023E-4	1.565E-3	1.599E-3	0.0921	1.389E-3	1.693E-5	1.209E-4
d_4	4.528E-3	6.791E-3	4.987E-3	0.0928	6.277E-3	4.570E-4	1.295E-3
d_5	0.0133	0.0150	0.0138	0.0928	0.0168	1.790E-3	4.425E-3
d_6	0.0259	0.0334	0.0379	0.0928	0.0643	4.581E-3	0.0123
d_7	0.0553	0.0604	0.0510	0.0928	0.0767	7.986E-3	0.0256
d_8	0.0682	0.0791	0.0668	0.0928	0.0767	0.0184	0.0402
d_9	0.1060	0.1125	0.0917	0.0928	0.1000	0.0314	0.0677
d_{10}	0.1085	0.1250	0.1155	0.1543	0.1387	0.0374	0.0702
d_{11}	0.1690	0.1429	0.1270	0.1543	0.1443	0.0541	0.0778
d_{12}	0.2425	0.1961	0.2132	0.1622	0.1581	0.0600	0.1005
d_{13}	0.2425	0.1961	0.2132	0.1961	0.1826	0.0693	0.1336
d_{14}	0.2425	0.2000	0.2132	0.2132	0.1961	0.0928	0.1336
d_{15}	0.2425	0.2000	0.2182	0.2132	0.2041	0.0953	0.1361
d_{16}	0.2425	0.2085	0.2294	0.2357	0.2236	0.1000	0.1414
d_{17}	0.2425	0.2425	0.2357	0.2673	0.2236	0.1291	0.1690
d_{18}	0.2500	0.2500	0.2500	0.2673	0.2236	0.1291	0.1690
d_{19}	0.2673	0.2500	0.2500	0.2673	0.2500	0.1471	0.1961
d_{20}	0.2673	0.2887	0.2673	0.2887	0.2500	0.1508	0.2041
d_{21}	0.2673	0.2887	0.2673	0.2887	0.3162	0.1667	0.2294
d_{22}	0.2887	0.2887	0.2774	0.2887	0.3162	0.1768	0.2294
d_{23}	0.2887	0.2887	0.2774	0.3162	0.3162	0.1890	0.2294
d_{24}	0.3015	0.2887	0.3015	0.3162	0.3162	0.1961	0.2294
d_{25}	0.3015	0.2887	0.3015	0.3162	0.3162	0.1961	0.2425
d_{26}	0.3015	0.2887	0.3015	0.3162	0.3162	0.1961	0.2425
d_{27}	0.3015	0.3015	0.3015	0.3162	0.3162	0.1961	0.2500
d_{28}	0.3015	0.3015	0.3333	0.3162	0.3162	0.2132	0.2673
d_{29}	0.3162	0.3015	0.3333	0.3162	0.3162	0.2236	0.2673
d_{30}	0.3162	0.3162	0.3333	0.3536	0.3162	0.2236	0.2673

Table 2: Distances between hyperplanes for some MRGs

	G8	G9	G10	G11
m	$2^{31}-19$	$2^{31}-19$	$(2^{31}-1)(2^{31}-2000169)$	$(2^{31}-85)(2^{31}-249)$
k	7	7	3	1
a_1	1975938786	1071064	2620007610006878699	1968402271571654650
a_2	875540239	0	4374377652968432818	
a_3	433188390	0	667476516358487852	
a_4	451413575	0		
a_5	1658907683	0		
a_6	1513645334	0		
a_7	1428037821	2113664		
S_2				0.66650
S_3				0.76439
S_4			0.75901	0.39148
S_5			0.77967	0.74850
S_6			0.75861	0.67560
S_7			0.76042	0.61124
S_8	0.73486	0.00696	0.74215	0.56812
$1/m$	4.6E-10	4.6E-10	4.6E-10	4.6E-10
d_2				6.5E-10
d_3				7.00E-7
d_4			1.1E-14	4.63E-5
d_5			6.6E-12	2.00E-4
d_6			4.7E-10	8.89E-4
d_7			9.80E-9	2.62E-3
d_8	6.57E-9	6.94E-7	9.55E-8	5.78E-3
d_9	5.91E-8	4.58E-6	6.00E-7	9.57E-3
d_{10}	2.87E-7	8.38E-6	2.24E-6	1.73E-2
d_{11}	1.08E-6	1.10E-5	8.41E-6	2.36E-2
d_{12}	3.85E-6	1.10E-5	2.66E-5	3.07E-2
d_{13}	9.29E-6	1.26E-5	4.68E-5	3.47E-2
d_{14}	1.99E-5	2.17E-5	1.05E-4	3.96E-2
d_{15}	4.17E-5	4.66E-5	1.60E-4	5.98E-2
d_{16}	7.63E-5	8.36E-5	2.68E-4	6.07E-2
d_{17}	1.33E-4	1.31E-4	4.26E-4	6.51E-2
d_{18}	2.77E-4	2.04E-4	7.05E-4	7.43E-2
d_{19}	2.95E-4	3.50E-4	1.03E-3	8.19E-2
d_{20}	4.62E-4	4.17E-4	1.32E-3	8.77E-2

a full period with $m = 2^{31} - 19$. For G9 there are the additional restrictions that a_1 and a_7 satisfy condition (5) and $a_i = 0$ for $2 \le i \le 6$. This m is the largest prime under 2^{31} such that $(m^7 - 1)/(m - 1)$ is also prime. The latter property facilitates the verification of condition (c) in the full-period conditions for an MRG. These two generators are taken from [73], where one can also find more details on the search and a precise definition of the selection criterion. It turns out that G9 has a very bad figure of merit S_8, and larger values of d_t than G8 for t slightly larger than 7. This is due to the restrictions $a_i = 0$ for $2 \le i \le 6$, under which the lower bound (9) is always much larger than d_t^* for $t = 8$. The distances between the hyperplanes for G9 are nevertheless much smaller than the corresponding values of any LCG of Table 1, so this generator is a clear improvement over those. G8 is better in terms of lattice structure, but also much more costly to run, because there are seven products modulo m to compute instead of two at each iteration of the recurrence. The other generators in this table are discussed later.

4.3.5 Lacunary Indices

Instead of constructing vectors of successive values as in (7), one can (more generally) construct vectors with values that are a fixed distance apart in the sequence, using *lacunary indices*. More specifically, let $I = \{i_1, i_2, \ldots, i_t\}$ be a given set of integers and define, for an MRG,

$$T_t(I) = \{(u_{i_1+n}, \ldots, u_{i_t+n}) \mid n \ge 0,\ s_0 = (x_0, \ldots, x_{k-1}) \in \mathbb{Z}_m^k\}.$$

Consider the lattice $L_t(I)$ spanned by $T_t(I)$ and $\mathbb{Z}^t$, and let $d_t(I)$ be the distance between the hyperplanes in this lattice. L'Ecuyer and Couture [77] show how to construct bases for such lattices, how to compute $d_t(I)$, and so on. The following provides "quick-and-dirty" lower bounds on $d_t(I)$ [13, 67]:

1. If I contains all the indices i such that $a_{k-i+1} \ne 0$, then

$$d_t(I) \ \ge \ \left(1 + \sum_{j=1}^{k} a_i^2\right)^{-1/2}. \tag{10}$$

In particular, if $x_n = (a_r x_{n-r} + a_k x_{n-k}) \bmod m$ and $I = \{0, k-r, k\}$, then $d_3(I) \geq (1 + a_r^2 + a_k^2)^{-1/2}$.

2. If m can be written as $m = \sum_{j=1}^{t} c_{i_j} a^{i_j}$ for some integers c_{i_j}, then

$$d_t(I) \geq \left(\sum_{j=1}^{t} c_{i_j}^2 \right)^{-1/2}. \tag{11}$$

As a special case of (10), consider the *lagged-Fibonacci generator*, based on a recurrence whose only two nonzero coefficients satisfy $a_r = \pm 1$ and $a_k = \pm 1$. In this case, for $I = \{0, k-r, k\}$, $d_3(I) \geq 1/\sqrt{3} \approx 0.577$. The set of all vectors $(u_n, u_{n+k-r}, u_{n+k})$ produced by such a generator lie in successive parallel planes that are at distance $1/\sqrt{3}$ to each other, and orthogonal to the vector $(1, 1, 1)$. Therefore, apart from the vector $(0, 0, 0)$, all other vectors of this form are contained in only two planes! Specific instances of this generator are the one proposed by Mitchell and Moore and recommended by Knuth [57], based on the recurrence $x_n = (x_{n-24} + x_{n-55}) \bmod 2^e$ for e equal to the computer's word length, as well as the `addrans` function in the SUN Unix library [117], based on $x_n = (x_{n-5} + x_{n-17}) \bmod 2^{24}$. These generators should not be used, at least not in their original form.

4.3.6 Combined LCGs and MRGs

Several authors advocated the idea of combining in some way different generators (e.g., two or three different LCGs), hoping that the composite generator will behave better than any of its components alone. See [10, 57, 60, 62, 87] and dozens of other references given there. Combination can provably increase the period length. Empirical tests show that it typically improves the statistical behavior as well. Some authors (e.g., [8, 46, 87]) have also given theoretical results which (on the surface) appear to "prove" that the output of a combined generator is "more random" than (or at least "as random" as) the output of each of its components. However, these theoretical results make sense only for random variables defined in a probability space setup. For (deterministic) pseudorandom sequences, they prove nothing and can be used only as heuristic arguments to support the idea of combination. To assess the quality of a specific combined generator, one should make a structural analysis of the combined generator itself, not only analyze the individual components and assume that combination will make things more random. This implies that the structural effect of the combination method must be well understood. Law and Kelton [60, Prob. 7.6] give an example where combination makes things worse.

The two most widely known combination methods are:

1. Shuffling one sequence with another or with itself.
2. Adding two or more integer sequences modulo some integer m_0, or adding sequences of real numbers in $[0, 1]$ modulo 1, or adding binary fractions bitwise modulo 2.

Shuffling one LCG with another can be accomplished as follows. Fill up a table of size d with the first d output values from the first LCG (suggested values of d go from 2 up to 128 or more). Then each time a random number is needed, generate an index $I \in \{1, \ldots, d\}$ using the $\log_2(d)$ most significant bits of the next output value from the *second* LCG, return (as output of the combined generator) the value stored in the table at position I, then replace this value by the next output value from the *first* LCG. Roughly, the first LCG produces the numbers and the second LCG changes the order of their occurrence. There are several variants of this shuffling scheme. In some of them, the same LCG that produces the numbers to fill up the table is also used to generate the values of I. A large number of empirical investigations performed over the past 30 years strongly support shuffling and many generators available in software libraries use it (e.g., [54, 110, 117]). However, it has two important drawbacks: (1) the effect of shuffling is not well-enough understood from the theoretical viewpoint, and (2) one does not know how to jump ahead quickly to an arbitrary point in the sequence of the combined generator.

The second class of combination method, by modular addition, is generally better understood theoretically. Moreover, jumping ahead in the composite sequence amounts to jumping ahead with each of the individual components, which we know how to do if the components are LCGs or MRGs.

Consider J MRGs evolving in parallel. The jth MRG is based on the recurrence

$$x_{j,n} = (a_{j,1}x_{j,n-1} + \cdots + a_{j,k}x_{j,n-k}) \bmod m_j,$$

for $j = 1, \ldots, J$. We assume that the moduli m_j are pairwise relatively prime and that each recurrence is purely periodic (has zero transient) with period length ρ_j. Let $\delta_1, \ldots, \delta_J$ be arbitrary integers such that for each j, δ_j and m_j have no common factor. Define the two combinations

$$z_n = \left(\sum_{j=1}^{J} \delta_j x_{j,n}\right) \bmod m_1 \qquad u_n = z_n/m_1 \tag{12}$$

and

$$w_n = \left(\sum_{j=1}^{J} \delta_j \frac{x_{j,n}}{m_j}\right) \bmod 1. \tag{13}$$

Let $k = \max(k_1, \ldots, k_J)$ and $m = \prod_{j=1}^{J} m_j$. The following results were proved in [80] for the case of MLCG components ($k = 1$) and in [65] for the more general case:

1. The sequences $\{u_n\}$ and $\{w_n\}$ both have period length $\rho = \mathrm{lcm}(\rho_1, \ldots, \rho_J)$ (the least common multiple of the period lengths of the components).

2. The w_n obey the recurrence

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod m; \qquad w_n = x_n/m, \tag{14}$$

   where the a_i can be computed by a formula given in [65] and do not depend on the δ_j.

3. One has $u_n = w_n + \epsilon_n$, with $\Delta^- \leq \epsilon_n \leq \Delta^+$, where Δ^- and Δ^+ can be computed as explained in [65] and are generally extremely small when the m_j are close to each other.

The combinations (12) and (13) can then be viewed as efficient ways to implement an MRG with very large modulus m. A structural analysis of the combination can be done by analyzing this MRG (e.g., its lattice structure). The MRG components can be chosen with only two nonzero coefficients a_{ij}, both satisfying condition (5), for ease of implementation, and the recurrence of the combination (14) can still have all of its coefficients nonzero and large. If each m_j is an odd prime and each MRG has maximal period length $\rho_j = m_j^{k_j} - 1$, each ρ_j is even, so $\rho \leq (m_1^{k_1} - 1) \cdots (m_J^{k_J} - 1)/2^{J-1}$ and this upper bound is attained if the $(m_j^{k_j} - 1)/2$ are pairwise relatively prime [65]. The combination (13) generalizes an idea of Wichmann and Hill [126], while (12) is a generalization of the combination method proposed by L'Ecuyer [61]. The latter combination somewhat scrambles the lattice structure because of the added "noise" ϵ_n.

Example 7 L'Ecuyer [65] proposes the following parameters and gives a computer code in the C language that implements (12). Take $J = 2$ components, $\delta_1 = -\delta_2 = 1$, $m_1 = 2^{31} - 1$, $m_2 = 2^{31} - 2000169$, $k_1 = k_2 = 3$, $(a_{1,1}, a_{1,2}, a_{1,3}) = (0, 63308, -183326)$, and $(a_{2,1}, a_{2,2}, a_{2,3}) = (86098, 0, -539608)$. Each component has period length $\rho_j = m_j^3 - 1$, and the combination has period length $\rho = \rho_1\rho_2/2 \approx 2^{185}$. The MRG (14) that corresponds to the combination is called G10 in Table 2, where distances between hyperplanes for the associated lattice are given. Generator G10 requires four modular products at each step of the recurrence, so it is slower than G9 but faster than G8. The combined MLCG originally proposed by L'Ecuyer [61] also has an approximating LCG called G11 in the table. Note that this combined generator was originally constructed on the basis of the lattice structure of the components only, *without* examining the lattice structure of the combination. Slightly better combinations of the same size have been constructed since this original proposal [80, 77]. Other combinations of different sizes are given in [68].

4.3.7 Matrix LCGs and MRGs

A natural way to generalize LCGs and MRGs is to consider linear recurrences for vectors, with matrix coefficients

$$X_n = (A_1 X_{n-1} + \cdots + A_k X_{n-k}) \bmod m, \tag{15}$$

where $A_1, \ldots, A_k$ are $L \times L$ matrices and each X_n is an L-dimensional vector of elements of $\mathbb{Z}_m$, which we denote by

$$X_n = \begin{pmatrix} x_{n,1} \\ \vdots \\ x_{n,L} \end{pmatrix}.$$

At each step, one can use each component of X_n to produce a uniform variate: $u_{nL+j-1} = x_{n,j}/m$. Niederreiter [105] introduced this generalization and calls it the *multiple recursive matrix method* for the generation of vectors. The recurrence (15) can also be written as a *matrix LCG* of the form $\boldsymbol{X}_n = \boldsymbol{A}\boldsymbol{X}_{n-1} \bmod m$, where

$$\boldsymbol{A} = \begin{pmatrix} 0 & I & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & I \\ A_k & A_{k-1} & \ldots & A_1 \end{pmatrix} \quad \text{and} \quad \boldsymbol{X}_n = \begin{pmatrix} X_n \\ X_{n+1} \\ \vdots \\ X_{n+k-1} \end{pmatrix} \tag{16}$$

are a matrix of dimension $kL \times kL$ and a vector of dimension kL, respectively (here I is the $L \times L$ identity matrix). This matrix notation applies to the MRG as well, with $L = 1$.

Is the matrix LCG more general than the MRG? Not much. If a k-dimensional vector X_n follows the recurrence $X_n = AX_{n-1} \bmod m$, where the $k \times k$ matrix A has a primitive characteristic polynomial $P(z) = z^k - a_1 z^{k-1} - \cdots - a_k$, then X_n also follows the recurrence [48, 62, 101]

$$X_n = (a_1 X_{n-1} + \cdots + a_k X_{n-k}) \bmod m \tag{17}$$

So each component of the vector X_n evolves according to (2). In other words, one simply has k copies of the same MRG sequence in parallel, usually with some shifting between those copies. This also applies to the matrix MRG (15), since it can be written as a matrix LCG of dimension kL and therefore corresponds to kL copies of the same MRG of order kL (and maximal period length $m^{kL}-1$). The difference with the single MRG (2) is that instead of taking successive values from a single sequence, one takes values from different copies of the same sequence, in a round-robin fashion. Observe also that when using (17), the dimension of X_n in this recurrence (i.e., the number of parallel copies) does not need to be equal to k.

4.3.8 Linear Recurrences with Carry

Consider a generator based on the following recurrence:

$$\begin{aligned} x_n &= (a_1 x_{n-1} + \cdots + a_k x_{n-k} + c_{n-1}) \bmod b, && (18)\\ c_n &= (a_1 x_{n-1} + \cdots + a_k x_{n-k} + c_{n-1}) \operatorname{div} b, && (19)\\ u_n &= x_n/b. \end{aligned}$$

where "div" denotes the integer division. For each n, $x_n \in \mathbb{Z}_b$, $c_n \in \mathbb{Z}$, and the state at step n is $s_n = (x_n, \ldots, x_{n+k-1}, c_n)$. As in [14, 16, 88], we call this a *multiply-with-carry* (MWC) generator. The idea was suggested in [58, 91]. The recurrence looks like that of an MRG, except that a *carry* c_n is propagated between the steps. What is the effect of this carry?

Assume that b is a power of 2, which is very nice form the implementation viewpoint. Define $a_0 = -1$,

$$m = \sum_{\ell=0}^{k} a_\ell b^\ell,$$

and let a be such that $ab \bmod m = 1$ (a is the inverse of b in arithmetic modulo m). Note that m could be either positive or negative, but for simplicity we now assume that $m > 0$. Consider the LCG:

$$z_n = a z_{n-1} \bmod m; \qquad w_n = z_n/m. \tag{20}$$

There is a close correspondence between the LCG (20) and the MWC generator, assuming that their initial states agree [16]. More specifically, if

$$w_n = \sum_{i=1}^{\infty} x_{n+i-1} b^{-i} \tag{21}$$

holds for $n = 0$, then it holds for all n. As a consequence, $|u_n - w_n| \le 1/b$ for all n. For example, if $b = 2^{32}$, then u_n and w_n are the same up to 32 bits of precision! The MWC generator can thus be viewed as just another way to implement (approximately) a LCG with huge modulus and period length. It also inherits from this LCG an approximate lattice structure, which can be analyzed as usual.

The LCG (20) is purely periodic, so each state z_n is *recurrent* (none is transient). On the other hand, the MWC has an infinite number of states (since we imposed no bound on c_n) and most of them turn out to be transient. How can one characterize the recurrent states? They are (essentially) the states s_0 that correspond to a given z_0 via (20)–(21). Couture and L'Ecuyer [16] give necessary and sufficient conditions for a state s_0 to be recurrent. In particular, if $a_\ell \ge 0$ for $\ell \ge 1$, all the recurrent states satisfy $0 \le c_n < a_1 + \cdots + a_k$. In view of this inequality, we want the a_ℓ to be small, for their sum to fit into a computer word. More specifically, one can impose $a_1 + \cdots + a_k \le b$. Now b is a nice upper bound on the c_n as well as on the x_n.

Since b is a power of 2, a is a quadratic residue and so cannot be primitive mod m. Therefore, the period length cannot reach $m - 1$ even if m is prime. But if $(m-1)/2$ is odd and 2 is primitive mod m (e.g., if $(m-1)/2$ is prime), then (20) has period length $\rho = (m-1)/2$.

Couture and L'Ecuyer [16] show that the lattice structure of the LCG (20) satisfies the following: In dimensions $t \le k$, the distances d_t do not depend on the parameters $a_1, \ldots, a_k$, but only on b, while in dimension $t = k+1$, the shortest vector in the dual lattice to L_t is $(a_0, \ldots, a_k)$, so that

$$d_t = (1 + a_1^2 + \cdots + a_k^2)^{-1/2}. \tag{22}$$

The distance d_{k+1} is then minimized if we put all the weight on one coefficient a_ℓ. It is also better to put more weight on a_k, to get a larger m. So one should choose a_k close to b, with $a_0 + \cdots + a_k \le b$. Marsaglia [88] proposed two specific parameter sets. They are analyzed in [16], where a better set of parameters in terms of the lattice structure of the LCG is also given.

Special cases of the MWC include the add-with-carry (AWC) and subtract-with-borrow (SWB) generators, originally proposed by Marsaglia and Zaman [91] and subsequently analyzed in [13, 122]. For the AWC, put $a_r = a_k = -a_0 = 1$ for $0 < r < k$ and all other a_ℓ equal to zero. This gives the simple recurrence

$$\begin{aligned} x_n &= (x_{n-r} + x_{n-k} + c_{n-1}) \bmod b, \\ c_n &= I[x_{n-r} + x_{n-k} + c_{n-1} \geq b], \end{aligned}$$

where I denotes the indicator function, equal to 1 if the bracketted inequality is true and to 0 otherwise. The SWB is similar, except that either a_r or a_k is -1 and the carry c_n is 0 or -1. The correspondence between AWC/SWB generators and LCGs was established in [122].

Equation (22) tells us very clearly that all AWC/SWB generators have a bad lattice structure in dimension $k+1$. A little more can be said when looking at the lacunary indices: For $I = \{0, r, k\}$, one has $d_3(I) = 1/\sqrt{3}$ and all vectors of the form (w_n, w_{n+r}, w_{n+k}) produced by the LCG (20) lie in only two planes in the three-dimensional unit cube, exactly as for the lagged-Fibonacci generators discussed in Section 4.3.5. Obviously, this is bad.

Perhaps one way to get around this problem is to take only k successive output values, then skip (say) ν values, take another k successive ones, skip another ν, and so on. Lüscher [85] has proposed such an approach, with specific values of ν for a specific SWB generator, with theoretical justification based on chaos theory. James [56] gives a Fortran implementation of Lüscher's generator. The system *Mathematica* uses a SWB generator ([127, p. 1019]), but the documentation does not specify if it skips values.

4.3.9 Digital Method: LFSR, GFSR, TGFSR, etc., and Their Combination

The MRG (2), matrix MRG (15), combined MRG (12), and MWC (18–19) have resolution $1/m$, $1/m$, $1/m_1$, and $1/b$, respectively. (The *resolution* is the largest number x such that all output values are multiples of x.) This could be seen as a limitation. To improve the resolution, one can simply take several successive x_n to construct each output value u_n. Consider the MRG. Choose two positive integers s and $L \leq k$, and redefine

$$u_n = \sum_{j=1}^{L} x_{ns+j-1} m^{-j}. \tag{23}$$

Call s the *step size* and L the *number of digits* in the m-adic expansion. The state at step n is now $s_n = (x_{ns}, \ldots, x_{ns+k-1})$. The output values u_n are multiples of m^{-L} instead of m^{-1}. This output sequence, usually with $L = s$, is called a *digital multistep sequence* [64, 102]. Taking $s > L$ means that $s - L$ values of the sequence $\{x_n\}$ are skipped at each step of (23). If the MRG sequence has period ρ and if s has no common factor with ρ, the sequence $\{u_n\}$ also has period ρ.

Now, it is no longer necessary for m to be large. A small m with large s and L can do as well. In particular, one can take $m = 2$. Then $\{x_n\}$ becomes a sequence of bits (zeros and ones) and the u_n are constructed by juxtaposing L successive bits from this sequence. This is called a *linear feedback shift register* (LFSR) or *Tausworthe generator* [41, 64, 102, 118], although the bits of each u_n are often filled in reverse order than in (23). An efficient computer code that implements the sequence (23), for the case where the recurrence has the form $x_n = (x_{n-r} + x_{n-k}) \bmod 2$ with $s \leq r$ and $2r > k$, can be found in [66, 120, 121]. For specialized jump-ahead algorithms, see [22, 66]. Unfortunately, such simple recurrences lead to LFSR generators with bad structural properties (see [11, 66, 97, 120] and other references therein). But combining several recurrences of this type can give good generators.

Consider J LFSR generators, where the jth one is based on a recurrence $\{x_{j,n}\}$ with primitive characteristic polynomial $P_j(z)$ of degree k_j (with binary coefficients), an m-adic expansion to L digits, and a step size s_j such that s_j and the period length $\rho_j = 2^{k_j} - 1$ have no common factor. Let $\{u_{j,n}\}$ be the output sequence of the jth generator and define u_n as the bitwise exclusive-or (i.e., bitwise addition modulo 2) of $u_{1,n}, \ldots, u_{j,n}$. If the polynomials $P_1(z), \ldots, P_J(z)$ are pairwise relatively prime (no pair of polynomials has a common factor), the period length ρ of the combined sequence $\{u_n\}$ is equal to the least common multiple of the individual periods $\rho_1, \ldots, \rho_J$. These ρ_j can be relatively prime, so it is possible here to have $\rho = \prod_{j=1}^{J} \rho_j$. The resulting combined generator is also exactly equivalent to a LFSR generator based on a recurrence with characteristic polynomial $P(z) = P_1(z) \cdots P_J(z)$. All of this is shown in [121], where specific combinations with two components are also suggested. For good combinations with more components, see [66]. Wang and Compagner [125] also suggested similar combinations, with much longer periods. They recommended constructing the combination so that the polynomial $P(z)$ has approximately half of its coefficients equal to 1. In a sense, the main justification for combined LFSR generators is the efficient implementation of a generator based on a (reducible) polynomial $P(z)$ with many nonzero coefficients.

The digital method can be applied to the matrix MRG (15) or to the parallel MRG (17) by making a digital expansion of the components of X_n (assumed to have dimension L):

$$u_n = \sum_{j=1}^{L} x_{n,j} m^{-j}. \tag{24}$$

The combination of (15) with (24) gives the *multiple recursive matrix method* of Niederreiter [103]. For the matrix LCG, L'Ecuyer [64] shows that if the shifts between the successive L copies of the sequence are all equal to some integer d having no common factor with the period length $\rho = m^k - 1$, the sequence (24) is exactly the same as the digital multistep sequence (23) with s equal to the inverse of d modulo m. The converse also holds. In other words, (23) and (24), with these conditions on the shifts, are basically two different implementations of the same generator. So one can be analyzed by analyzing the other, and vice versa. If one uses the implementation (24), one must be careful with the initialization of $X_0, \ldots, X_{k-1}$ in (17) to maintain the correspondence: The shift between the states $(x_{0,j}, \ldots, x_{k-1,j})$ and $(x_{0,j+1}, \ldots, x_{k-1,j+1})$ in the MRG sequence must be equal to the proper value d for all j.

The implementation (24) requires more memory than (23), but may give a faster generator. An important instance of this is the *generalized feedback shift register* (GFSR) *generator* [43, 84, 123] which we now describe. Take $m = 2$ and L equal to the computer's word length. The recurrence (17) can then be computed by a bitwise exclusive-or of the X_{n-j} for which $a_j = 1$. In particular, if the MRG recurrence has only two nonzero coefficients, say a_k and a_r, we obtain

$$X_n = X_{n-r} \oplus X_{n-k},$$

where $\oplus$ denotes the bitwise exclusive-or. The output is then constructed via the binary fractional expansion (24). This GFSR can be viewed as a different way to implement a LFSR generator, provided that it is initialized accordingly, and the structural properties of the GFSR can then be analyzed by analyzing those of the corresponding LFSR generator [44, 64].

For the recurrence (17), we need to memorize kL integers in $\mathbb{Z}_m$. With this memory size, one should expect a period length close to m^{kL}, but the actual period length cannot exceed $m^k - 1$. A big waste! Observe that (17) is a special case of (15), with $A_i = a_i I$. An interesting idea is to "twist" the recurrence (17) slightly so that each $a_i I$ is replaced by a matrix A_i such that the corresponding recurrence (15) has full period length $m^{kL} - 1$ while its implementation remains essentially as fast as (17). Matsumoto and Kurita [95, 96] proposed a specific way to do this for GFSR generators and called the resulting generators *twisted* GFSR (TGFSR). Their second paper and [98, 120] point out some defects in the generators proposed in their first paper, proposes better specific generators, and give nice computer codes in C. Investigations are currently made to find other twists with good properties. The multiple recursive matrix method of [103] is a generalization of these ideas.

4.3.10 Equidistribution Properties for the Digital Method

Suppose that we partition the unit hypercube $[0,1)^t$ into $m^{t\ell}$ cubic cells of equal size. This is called a (t,ℓ)-*equidissection in base* m. A set of points is said to be (t,ℓ)-equidistributed if each cell contains the same number of points from that set. If the set contains m^k points, the (t,ℓ)-equidistribution is possible only for $\ell \le \lfloor k/t \rfloor$. For a given digital multistep sequence, let

$$T_t = \{\boldsymbol{u}_0 = (u_0, \ldots, u_{t-1}) \mid (x_0, \ldots, x_{k-1}) \in \mathbb{Z}_m^k\} \tag{25}$$

(where repeated points are counted as many times as they appear in T_t) and $\ell_t = \min(L, \lfloor k/t \rfloor)$. If the set T_t is (t, ℓ_t)-equidistributed for all $t \le k$, we call it a *maximally equidistributed* (ME) set and say that the generator is ME. If it has the additional property that for all t, for $\ell_t < \ell \le L$, no cell of the (t, ℓ)-equidissection contains more than one point, we also call it collision-free (CF). ME-CF generators have their sets of points T_t very evenly distributed in the unit hypercube, in all dimensions t.

Full-period LFSR generators are all $(\lfloor k/s \rfloor, s)$-equidistributed. Full-period GFSR generators are all $(k, 1)$-equidistributed, but their (k, ℓ)-equidistribution for $\ell > 1$ depends on the initial state (i.e., on the shifts between the different copies of the MRG). Fushimi and Tezuka [45] give a necessary and sufficient condition on this initial state for (t, L)-equidistribution, for $t = \lfloor k/L \rfloor$. The condition says that the tL bits $(x_{0,1}, \ldots, x_{0,L}, \ldots, x_{t-1,1}, \ldots, x_{t-1,L})$ must be independent, in the sense that the $tL \times k$ matrix which expresses them as a linear transformation of $(x_{0,1}, \ldots, x_{k-1,1})$ has (full) rank tL. Fushimi [44] gives an initialization procedure satisfying this condition.

Couture et al. [17] show how the (t, ℓ)-equidistribution of simple and combined LFSR generators can be analyzed via the lattice structure of an equivalent LCG in a space of formal series. A different (simpler) approach is taken in [66]: Check if the matrix that expresses the first ℓ bits of $\boldsymbol{u}_n$ as a linear transformation of $(x_0, \ldots, x_{k-1})$ has full rank. This is a necessary and sufficient condition for (t, ℓ)-equidistribution.

An ME LFSR generator based on the recurrence $x_n = (x_{n-607} + x_{n-273}) \bmod 2$, with $s = 512$ and $L = 23$, is given in [123]. But as stated previously, only two nonzero coefficients for the recurrence is much too few. L'Ecuyer [66, 70] gives the results of computer searches for ME and ME-CF combined LFSR generators with $J = 2, 3, 4, 5$ components, as described in subSection 4.3.9. Each search was made within a class with each component j based on a characteristic trinomial $P_j(z) = z^{k_j} - z^{r_j} - 1$, with $L = 32$ or $L = 64$, and step size s_j such that $s_j \le r_j$ and $2r_j > k_j$. The period length is $\rho = (2^{k_1} - 1) \cdots (2^{k_J} - 1)$ in most cases, sometimes slightly smaller. The searches were for good parameters r_j and s_j. We summarize here a few examples of search results. For more details, as well as specific implementations in the C language, see [66, 70].

Example 8

(a) For $J = 2$, $k_1 = 31$, and $k_2 = 29$, there are 2565 parameter sets that satisfy the conditions above. None of these combinations is ME. Specific combinations which are nearly ME, within this same class, can be found in [121].

(b) Let $J = 3$, $k_1 = 31$, $k_2 = 29$, and $k_3 = 28$. In an exhaustive search among 82080 possibilities satisfying our conditions within this class, 19 ME combinations were found, and 3 of them are also CF.

(c) Let $J = 4$, $k_1 = 31$, $k_2 = 29$, $k_3 = 28$, and $k_4 = 25$. Here, in an exhaustive search among 3283200 possibilities, we found 26195 ME combinations, and 4744 of them also CF.

These results illustrate the fact that ME combinations are much easier to find as J increases. This appears to be due to more possibilities to "fill up" the coefficients of $P(z)$ when it is the product of more trinomials. Since GFSR generators can be viewed as a way to implement fast LFSR generators, these search methods and results can be used as well to find good combined GFSRs, where the combination is defined by a bitwise exclusive-or as in the LFSR case.

One may strenghten the notion of (t, ℓ)-equidistribution as follows: Instead of looking only at equidissections comprised of cubic volume elements of identical sizes, look at more general partitions. Such a stronger notion is that of a (q, k, t)-*net in base* m, where there should be the same number of points in each box for *any* partition of the unit hypercube into rectangular boxes of identical shape and equal volume m^{q-k}, with the length of each side of the box equal to a multiple of $1/m$. Niederreiter [102] defines a figure of merit $r^{(t)}$ such that for all $t > \lfloor k/L \rfloor$, the m^k points of T_t for (23) form a (q, k, t)-net in base m with $q = k - r^{(t)}$. A problem with $r^{(t)}$ is the difficulty to compute it for medium and large t (say, $t > 8$).

4.4 NONLINEAR METHODS

An obvious way to remove the linear (and perhaps too regular) structure is to use a *nonlinear* transformation. There are basically two classes of approaches:

1. Keep the transition function T linear, but use a nonlinear transformation G to produce the output.

2. Use a nonlinear transition function T.

Several types of nonlinear generators have been proposed over the last decade or so, and an impressive volume of theoretical results have been obtained for them. See, for example, [31, 34, 59, 78, 102, 104] and other references given there. Here, we give a brief overview of this rapidly developing area.

Nonlinear generators avoid lattice structures. Typically, no t-dimensional hyperplane contains more than t overlapping t-tuples of successive values. More important, their output behaves much like "truly" random numbers, even over the entire period, with respect to discrepancy. Roughly, there are lower and upper bounds on their discrepancy (or in some cases on the average discrepancy over a certain set of parameters) whose asymptotic order (as the period length increases to infinity) is the same as that of an IID $U(0,1)$ sequence of random variables. They have also succeeded quite well in empirical tests performed so far [49]. Fast implementations with specific well-tested parameters are still under development, although several generic implementations are already available [49, 71].

4.4.1 Inversive Congruential Generators

To construct a nonlinear generator with long period, a first idea is simply to add a nonlinear twist to the output of a known generator. For example, take a full-period MRG with prime modulus m and replace the output function $u_n = x_n/m$ by

$$z_n = (\tilde{x}_{n+1}\tilde{x}_n^{-1}) \bmod m \qquad \text{and} \qquad u_n = z_n/m, \tag{26}$$

where $\tilde{x}_i$ denotes the ith nonzero value in the sequence $\{x_n\}$ and $\tilde{x}_n^{-1}$ is the inverse of $\tilde{x}_n$ modulo m. (The zero values are skipped because they have no inverse.) For $x_n \neq 0$, its inverse x_n^{-1} can be computed by the formula $x_n^{-1} = x_n^{m-2} \bmod m$, with $O(\log m)$ multiplications modulo m. The sequence $\{z_n\}$ has period m^{k-1}, under conditions given in [31, 102]. This class of generators was introduced and first studied in [28, 27, 30]. For $k = 2$, (26) is equivalent to the recurrence

$$z_n = \begin{cases} (a_1 + a_2 z_{n-1}^{-1}) \bmod m & \text{if } z_{n-1} \neq 0; \\ a_1 & \text{if } z_{n-1} = 0, \end{cases} \tag{27}$$

where a_1 and a_2 are the MRG coefficients.

A more direct approach is the *explicit inversive congruential* method of [32], defined as follows. Let $x_n = an + c$ for $n \geq 0$, where $a \neq 0$ and c are in $\mathbb{Z}_m$ and m is prime. Then, define

$$z_n = x_n^{-1} = (an + c)^{m-2} \bmod m \qquad \text{and} \qquad u_n = z_n/m. \tag{28}$$

This sequence has period $\rho = m$. According to [34], this family of generators seems to enjoy the most favorable properties among the currently proposed inversive and quadratic families. As a simple illustrative example, take $m = 2^{31} - 1$ and $a = c = 1$. (However, at the moment, we are not in a position to recommend these particular parameters nor any other specific ones.)

Inversive congruential generators with power-of-2 moduli have also been studied [30, 31, 35]. However, they have have more regular structures than those based on prime moduli [31, 34]. Their low-order bits have the same short period lengths as for the LCGs. The idea of combined generators, discussed earlier for the linear case, also applies to nonlinear generators and offers some computational advantages. Huber [52] and Eichenauer-Herrmann [33] introduced and analyzed the following method. Take J inversive generators as in (27), with distinct prime moduli $m_1, \ldots, m_J$, all larger than 4, and full period length $\rho_j = m_j$. For each generator j, let $z_{j,n}$ be the state at step n and let $u_{j,n} = z_{j,n}/m_j$. The output at step n is defined by the following combination:

$$u_n = (u_{1,n} + \cdots + u_{J,n}) \bmod 1.$$

The sequence $\{u_n\}$ turns out to be equivalent to the output of an inversive generator (27) with modulus $m = m_1 \cdots m_J$ and period length $\rho = m$. Conceptually, this is pretty similar to the combined LCGs and MRGs discussed previously, and provides a convenient way to implement an inversive generator with large modulus m. Eichenauer-Herrmann [33] shows that this type of generator has favorable asymptotic discrepancy properties, much like (26)–(28).

4.4.2 Quadratic Congruential Generators

Suppose that the transformation T is *quadratic* instead of linear. Consider the recurrence

$$x_n = (ax_{n-1}^2 + bx_{n-1} + c) \bmod m,$$

where $a, b, c \in \mathbb{Z}_m$ and $x_n \in \mathbb{Z}_m$ for each n. This is studied in [29, 37, 57, 102]. If m is a power of 2, this generator has full period ($\rho = m$) if and only if a is even, $(b - a) \bmod 4 = 1$, and c is odd. Its t-dimensional points turn out to lie on a union of grids. Also, the discrepancy tends to be too large. Our usual caveat against power-of-2 moduli applies again.

4.4.3 BBS and Other Cryptographic Generators

The BBS generator, explained in Section 4.2, is conjectured to be polynomial-time perfect. This means that for a large enough size k, a BBS generator with properly (randomly) chosen parameters is practically certain to behave very well from the statistical point of view. However, it is not clear how large k must be and how K can be chosen in practice for the generator to be really safe. The speed of the generator slows down with k, since at each step we must square a $2k$-bit integer modulo another $2k$-bit integer. An implementation based on fast modular multiplication is proposed by Moreau [99].

Other classes of generators, conjectured to be polynomial-time perfect, have been proposed. From empirical experiments, they have appeared no better than the BBS. See [5, 59, 78] for overviews and discussions. An interesting idea, pursued for instance in

[1], is to combine a slow but cryptographically strong generator (e.g., a polynomial-time perfect one) with a fast (but unsecure) one. The slow generator is used sparingly, mostly in a preprocessing step. The result is an interesting compromise between speed, size, and security. In [1], it is also suggested to use a block cipher encryption algorithm for the slow generator. These authors actually use triple-DES (three passes over the well-known data encryption standard algorithm, with three different keys), combined with a linear hashing function defined by a matrix. The keys and the hashing matrix must be (truly) random. Their fast generator is implemented with a six-regular expander graph (see their paper for more details).

4.5 EMPIRICAL STATISTICAL TESTING

Statistical testing of random number generators is indeed a very empirical and heuristic activity. The main idea is to seek situations where the behavior of some function of the generator's output is significantly different than the normal or expected behavior of the same function applied to a sequence of IID uniform random variables.

Example 9 As a simple illustration, suppose that one generates n random numbers from a generator whose output is supposed to imitate IID $U(0,1)$ random variables. Let T be the number of values that turn out to be below $1/2$, among those n. For large n, T should normally be not too far from $n/2$. In fact, one should expect T to behave like a binomial random variable with parameters $(n, 1/2)$. So if one repeats this experiment several times (e.g., generating N values of T), the distribution of the values of T obtained should resemble that of the binomial distribution (and the normal distribution with mean $n/2$ and standard deviation $\sqrt{n}/2$ for large n). If $N = 100$ and $n = 10000$, the mean and standard deviation are 5000 and 50, respectively. With these parameters, if one observes, for instance, that 12 values of T are less than 4800, or that 98 values of T out of 100 are less than 5000, one would readily conclude that something is wrong with the generator. On the other hand, if the values of T behave as expected, one may conclude that the generator seems to reproduce the correct behavior *for this* *particular statistic* T (and for this particular sample size). But nothing prevents *other* statistics than this T to behave wrongly.

4.5.1 General Setup

Define the null hypothesis H_0 as: "The generator's output is a sequence of IID $U(0,1)$ random variables". Formally, this hypothesis is false, since the sequence is periodic and usually deterministic (except parhaps for the seed). But if this cannot be detected by reasonable statistical tests, one may assume that H_0 holds anyway. In fact, what really counts in the end is that the statistics of interest in a given simulation have (sample) distributions close enough to their theoretical ones.

A statistical test for H_0 can be defined by any function T of a finite number of $U(0,1)$ random variables, for which the distribution under H_0 is known or can be approximated well enough. The random variable T is called the *test statistic*. The statistical test tries to find empirical evidence against H_0.

When applying a statistical test to a random number generator, a *single-level* procedure computes the value of T, say t_1, then computes the p-value

$$\delta_1 = P[T > t_1 \mid H_0],$$

and, in the case of a two-sided test, rejects H_0 if δ_1 is too close to either 0 or 1. A single-sided test will reject only of δ_1 is too close to 0, or only if it is too close to 1. The choice of rejection area depends on what the test aims to detect. Under H_0, δ_1 is a $U(0,1)$ random variable.

A *two-level* test obtains (say) N "independent" copies of T, denoted $T_1, \ldots, T_N$, and computes their empirical distribution $\hat{F}_N$. This empirical distribution is then compared to the theoretical distribution of T under H_0, say F, via a standard goodness-of-fit test, such as the Kolmogorov–Smirnov (KS) or Anderson–Darling tests [25, 115]. One version of the KS goodness-of-fit test uses the statistic

$$D_N = \sup_{-\infty < x < \infty} |\hat{F}_N(x) - F(x)|,$$

for which an approximation of the distribution under H_0 is available, assuming that the distribution F is continuous [25]. Once the value d_N of the statistic D_N is known, one computes the p-value of the test, defined as

$$\delta_2 = P[D_N > d_N \mid H_0],$$

which is again a $U(0,1)$ random variable under H_0. Here one would reject H_0 if δ_2 is too close to 0.

Choosing $N = 1$ yields a single-level test. For a given test and a fixed computing budget, the question arises of what is best: To choose a small N (e.g., $N = 1$) and base the test statistic T on a large sample size, or the opposite? There is no universal winner. It depends on the test and on the alternative hypothesis. The rationale for two-level testing is to test the sequence not only globally, but also locally, by looking at the distribution of values of T over shorter subsequences [57]. In most cases, when testing random number generators, $N = 1$ turns out to be the best choice because the same regularities or defects of the generators tend to repeat themselves over all long-enough subsequences. But it also happens for certain tests that the cost of computing T increases faster than linearly with the sample size, and this gives another argument for choosing $N > 1$.

In statistical analyses where a limited amount of data is available, it is common practice to fix some significance level α in advance and reject H_0 when and only when the p-value is below α. Popular values of α are 0.05 and 0.01 (mainly for historical reasons). When testing random number generators, one can always produce an arbitrary amount of data to make the test more powerful and come up with a clean-cut decision when suspicious p-values occur. We would thus recommend the following strategy. If the outcome is clear, for example if the p-value is less than 10^{-10}, reject H_0. Otherwise, if the p-value is suspicious (0.005, for example), then increase the sample size or repeat the test with other segments of the sequence. In most cases, either suspicion will disappear or clear evidence against H_0 will show up rapidly.

When H_0 is not rejected, this somewhat improves confidence in the generator but never proves that it will always behave correctly. It may well be that the next test

T to be designed will be the one that catches the generator. Generally speaking, the more extensive and varied is the set of tests that a given generator has passed, the more faith we have in the generator. For still better confidence, it is always a good idea to run important simulations twice (or more), using random number generators of totally different types.

4.5.2 Available Batteries of Tests

The statistical tests described by Knuth [57] have long been considered the "standard" tests for random number generators. A Fortran implementation of (roughly) this set of tests is given in the package TESTRAND [24]. A newer battery of tests is DIEHARD, designed by Marsaglia [87, 89]. It contains more stringent tests than those in [57], in the sense that more generators tend to fail some of the tests. An extensive testing package called `TestU01` [71], that implements most of the tests proposed so far, as well as several classes of generators implemented in generic form, is under development. References to other statistical tests applied to random number generators can be found in [63, 64, 71, 75, 74, 69, 79, 116].

Simply testing uniformity, or pair correlations, is far from enough. Good tests are designed to catch higher-order correlation properties or geometric patterns of the successive numbers. Such patterns can easily show up in certain classes of applications [39, 49, 75]. Which are the best tests? No one can really answer this question. If the generator is to be used to estimate the expectation of some random variable T by generating replicates of T, the best test would be the one based on T as a statistic. But this is impractical, since if one knew the distribution of T, one would not use simulation to estimate its mean. Ideally, a good test for this kind of application should be based on a statistic T' whose distribution is known and resembles that of T. But such a test is rarely easily available. Moreover, only the user can apply it. When designing a general purpose generator, one has no idea of what kind of random variable interests the user. So, the best the designer can do (after the generator has been properly designed) is to apply a wide variety of tests that tend to detect defects of different natures.

4.5.3 Two Examples of Empirical Tests

For a short illustration, we now apply two statistical tests to some of the random number generators discussed previously. The first test is a variant of the well-know *serial test* and the second one is a *close-pairs* test. More details about these tests, as well as refined variants, can be found in [57, 74, 75, 79].

Both tests generate n nonoverlapping vectors in the t-dimensional unit cube $[0,1)^t$. That is, they produce the point set:

$$P_t = \{\boldsymbol{U}_i = (U_{t(i-1)}, \ldots, U_{ti-1}),\ i = 1, \ldots, n\},$$

where $U_0, U_1, \ldots$ is the generator's output. Under H_0, P_t contains n IID random vectors uniformly distributed over the unit hypercube.

For the serial test, we construct a (t, ℓ)-equidissection in base 2 of the hypercube (see Section 4.3.10), and compute how many points fall in each of the $k = 2^{t\ell}$ cells. More specifically, let X_j be the number of points $\boldsymbol{U}_i$ falling in cell j, for $j = 1, \ldots, k$, and define the chi-square statistic

$$X^2 = \sum_{j=1}^{k} \frac{(X_j - n/k)^2}{n/k}. \tag{29}$$

Under H_0, the exact mean and variance of X^2 are $\mu = E[X^2] = k - 1$ and $\sigma^2 = \mathrm{Var}[X^2] = 2(k-1)(n-1)/n$, respectively. Moreover, if $n \to \infty$ for fixed k, X^2 converges in distribution to a chi-square random variable with $k-1$ degrees of freedom, whereas if $n \to \infty$ and $k \to \infty$ simultaneously so that $n/k \to \gamma$ for some constant γ, $(X^2 - \mu)/\sigma$ converges in distribution to a $N(0,1)$ (a standard normal) random variable. Most authors use a chi-square approximation to the distribution of X^2, with $n/k \geq 5$ (say) and very large n. But one can also take $k \gg n$ and use the normal approximation, as in the forthcoming numerical illustration.

For the close-pairs test, let $D_{n,i,j}$ be the Euclidean distance between the points $\boldsymbol{U}_j$ and $\boldsymbol{U}_i$ in the unit *torus*, i.e., where the opposite faces of the hypercube are identified so that points facing each other on opposite sides become close to each other. For $s \geq 0$, let $Y_n(s)$ be the number of distinct pairs of points $i < j$ such $D^t_{n,i,j} V_t n(n-1)/2 \le s$, where V_t is the volume of a ball of radius 1 in the t-dimensional real space. Under H_0, for any constant $s_1 > 0$, as $n \to \infty$, the process $\{Y_n(s),\, 0 \le s \le s_1\}$ converges weakly to a Poisson process with unit rate. Let $0 = T_{n,0} \le T_{n,1} \le T_{n,2} \le \cdots$ be the jump times of the process Y_n, and let $W_{n,i} = 1 - \exp[-(T_{n,i} - T_{n,i-1})]$. For a fixed integer $m > 0$ and large enough n, the random variables $W_{n,1}, \ldots, W_{n,m}$ are approximately IID $U(0,1)$ under H_0. To compare their empirical distribution to the uniform, one can compute, for example, the Anderson–Darling statistic

$$A^2_m \;=\; -m - \frac{1}{m}\sum_{i=1}^{m}\left\{(2i-1)\ln(W_{(n,i)}) + (2m+1-2i)\ln(1-W_{(n,i)})\right\},$$

and reject H_0 if the p-value is too small (i.e., if A^2_m is too large).

These tests have been applied to the generators G1 to G11 in Tables 4.1 and 4.2. We took $N = 1$ and dimension $t = 3$. We applied two instances of the serial test, one named ST1, with $n = 2^{20}$ and $\ell = 9$, which gives $k = 2^{27}$ and $n/k = 1/128$, and the second one named ST2, with $n = 2^{22}$ and $\ell = 10$, so $k = 2^{30}$ and $n/k = 1/256$. For the close-pairs (CP) test, we took $n = 2^{18}$ and $m = 32$. In each case, $3n$ random numbers were used, and this value is much smaller than the period length of the generators tested. For all generators, at the beginning of the first test, we used the initial seed 12345 when a single integer was needed and the vector $(12345, \ldots, 12345)$ when a vector was needed. The seed was not reset between the tests. Table 3 gives the p-values of these tests for G1 to G5. For G6 to G11, all p-values remained inside the interval $(0.01, 0.99)$.

For the serial test, the p-values that are too close to 1 (e.g., ST1 and ST2 for G1) indicate that the n points are too evenly distributed among the k cells compared to what one would expect from random points (X^2 is too small). On the other hand, the very small p-values indicate that the points tend to go significantly more often in certain cells than in others (X^2 is too large). The p-values less than 10^{-15} for the CP test stem from the fact that the jumps of the process Y_n tend to be clustered (and often superposed), because there are often equalities (or almost) among the small $D_{n,i,j}$'s, due to the lattice structure of the generator [75, 112]. This implies that several $W_{n,i}$ are very close to zero, and the Anderson-Darling statistic is especially sensitive for detecting this type of

Table 3: The p-values of two empirical tests applied to Generators G1 to G11.

Generator	ST1	ST2	CP
G1	$1 - 9.97 \times 10^{-6}$	$> 1 - 10^{-15}$	$< 10^{-15}$
G2	0.365	$< 10^{-15}$	$< 10^{-15}$
G3	$1 - 2.19 \times 10^{-4}$	$< 10^{-15}$	$< 10^{-15}$
G4	$< 10^{-15}$	$< 10^{-15}$	$< 10^{-15}$
G5	0.950	$> 1 - 10^{-15}$	$< 10^{-15}$

problem. As a general rule of thumb, *all* LCGs and MRGs, whatever be the quality of their lattice structure, fail spectacularly this close-pairs test with $N = 1$ and $m = 32$ when n exceeds the square root of the period length [75].

G6 and G7 pass these tests, but will soon fail both tests if we increase the sample size. For G8 to G11, on the other hand, the sample size required for clear failure is so large that the test becomes too long to run in reasonable time. This is especially true for G8 and G10.

One could raise the issue of whether these tests are really relevant. As mentioned in the previous subsection, the relevant test statistics are those that behave similarly as the random variable of interest to the user. So, relevance depends on the application. For simulations that deal with random points in space, the close-pairs test could be relevant. Such simulations are performed, for example, to estimate the (unknown) distribution of certain random variables in spatial statistics [19]. As an illustration, suppose one wishes to estimate the distribution of $\min_{i,j} D_{n,i,j}$ for some fixed n, by Monte Carlo simulation. For this purpose I would not trust the generators G1 to G5. The effect of failing the serial or close-pairs test in general is unclear. In many cases, if not so many random numbers are used and if the application does not interact constructively with the structure of the point set produced by the generator, no bad effect will show up. On the other hand, simulations using more than, say, 2^{32} random numbers are becoming increasingly common. Clearly, G1 to G5 and all other generators of that size are unsuitable for such simulations.

4.5.4 Empirical Testing: Summary

Experience from years of empirical testing with different kinds of tests and different generator families provides certain guidelines [49, 63, 75, 74, 69, 89, 81], Some of these guidelines are summarized in the following remarks.

1. Generators with period length less than 2^{32} (say) can now be considered as "baby toys" and should not be used in general software packages. In particular, all LCGs of that size fail spectacularly several tests that run in a reasonably short time and use much less random numbers than the period length.

2. LCGs with power-of-2 moduli are easier to crack than those with prime moduli, especially if we look at lower-order bits.

3. LFSRs and GFSRs based on primitive trinomials, or lagged-Fibonacci and AWC/SWB generators, whose structure is too simple in moderately large dimension, also fail several simple tests.

4. Combined generators with long periods and good structural properties do well in the tests. When a large fraction of the period length is used, nonlinear inversive generators with prime modulus do better than the linear ones.

5. In general, generators with good theoretical figures of merit (e.g., good lattice structure or good equidistribution over the entire period, when only a small fraction of the period is used) behave better in the tests. As a crude general rule, generators based on more complicated recurrences (e.g., combined generators) and good theoretical properties perform better.

4.6 PRACTICAL RANDOM NUMBER PACKAGES

4.6.1 Recommended Implementations

As stated previously, no random number generator can be guaranteed against all possible defects. However, there are generators with fairly good theoretical support, that have been extensively tested, and for which computer codes are available. We now give references to such implementations. Some of them are already mentioned earlier. We do not reproduce the computer codes here, but the user can easily find them from the references. More references and pointers can be found from the pages `http://www.iro.umontreal.ca/~lecuyer` and `http://random.mat.sbg.ac.at` on the World Wide Web.

Computer implementations that this author can suggest for the moment include those of the MRGs given in [73], the combined MRGs given in [65, 68], the combined Tausworthe generators given in [66, 70], the twisted GFSRs given in [96, 98], and perhaps the RANLUX code of [56].

4.6.2 Multigenerator Packages with Jump-Ahead Facilities

Good simulation languages usually offer many (virtual) random number generators, often numbered 1, 2, 3, ... In most cases this is the same generator but starting with different seeds, widely spaced in the sequence. L'Ecuyer and Côté [76] have constructed a package with 32 generators (which can be easily extended to 1024). Each generator is in fact based on the same recurrence (a combined LCG of period length near 2^{61}), with seeds spaced 2^{50} values apart. Moreover, each subsequence of 2^{50} values is split further into 2^{20} segments of length 2^{30}. A simple procedure call permits one to have any of the generators jump ahead to the beginning of its next segment, or its current segment, or to the beginning of its first segment. The user can also set the initial seed of the first generator to any admissible value (a pair of positive integers) and all other initial seeds are automatically recalculated so that they remain 2^{50} values apart. This is implemented with efficient jump-ahead tools. A boolean switch can also make any generator produce antithetic variates if desired.

To illustrate the utility of such a package, suppose that simulation is used to compare two similar systems using common random numbers, with n simulation runs for each system. To ensure proper synchronization, one would typically assign different generators to different streams of random numbers required by the simulation (e.g., in a queueing network, one stream for the interarrival times, one stream for the service times at each node, one stream for routing decisions, etc.), and make sure that for each run, each generator starts at the same seed and produces the same sequence of numbers for the two systems. Without appropriate tools, this may require tricky programming, because the two systems do not necessarily use the same number of random numbers in a given run. But with the package in [76], one can simply assign each run to a segment number. With the first system, use the initial seed for the first run, and before each new run, advance each generator to the beginning of the next segment. After the nth run, reset the generators to their initial seeds and do the same for the second system.

The number and length of segments in the package of [76] are now deemed too small for current and future needs. A similar package based on a combined LCG with period length near 2^{121} in given in [72], and other systems of this type, based on generators with much larger periods, are under development. In some of those packages, generators can be seen as objects that can be created by the user as needed, in practically unlimited number.

When a generator's sequence is cut into subsequences spaced, say, ν values apart as we just described, to provide for multiple generators running in parallel, one must analyze and test the vectors of nonsuccessive output values (with lacunary indices; see Section 4.3.5) spaced ν values apart. For LCGs and MRGs, for example, the lattice structure can be analyzed with such lacunary indices. See [38, 77] for more details and numerical examples.

4.6.3 Generators for Parallel Computers

Another situation where multiple random number generators are needed is for simulation on parallel processors. The same approach can be taken: Partition the sequence of a single random number generator with very long period into disjoint subsequences and use a different subsequence on each processor. So the same packages that provide multiple generators for sequential computers can be used to provide generators for parallel processors. Other approaches, such as using completely different generators on the different processors or using the same type of generator with different parameters (e.g., changing the additive term or the multiplier in a LCG), have been proposed but appear much less convenient and sometimes dangerous [62, 64]. For different ideas and surveys on parallel generators, the reader can consult [2, 9, 22, 93, 109].

ACKNOWLEDGMENTS

This work has been supported by NSERC-Canada Grant ODGP0110050 and SMF0169893, and FCAR-Québec Grant 93ER1654. Thanks to Christos Alexopoulos, Jerry Banks, Raymond Couture, Hannes Leeb, Thierry Moreau, and Richard Simard for their helpful comments.

REFERENCES

1. Aiello, W., S. Rajagopalan and R. Venkatesan (1996). Design of practical and provably good random number generators. Manuscript (contact `venkie@bellcore.com`).
2. Anderson, S. L. (1990). Random number generators on vector supercomputers and other advanced architecture. *SIAM Review*, Vol. 32, pp. 221–251.
3. Atkinson, A. C. (1980). Tests of pseudo-random numbers. *Applied Statistics*, Vol. 29, pp. 164–171.
4. Blum, L., M. Blum and M. Schub (1986). A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing*, Vol. 15, No. 2, pp. 364–383.
5. Boucher, M. (1994). La génération pseudo-aléatoire cryptographiquement sécuritaire et ses considérations pratiques. Master's thesis, Département d'I.R.O., Université de Montréal.
6. Brassard, G. (1988). *Modern Cryptology - A Tutorial*, volume 325 of *Lecture Notes in Computer Science*. Springer Verlag.
7. Bratley, P., B. L. Fox and L. E. Schrage (1987). *A Guide to Simulation*, second edition. Springer-Verlag, New York.
8. Brown, M. and H. Solomon (1979). On combining pseudorandom number generators. *Annals of Statistics*, Vol. 1, pp. 691–695.

9. Chen, J. and P. Whitlock (1995). Implementation of a distributed pseudorandom number generator. In H. Niederreiter and P. J.-S. Shiue, editors, *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, number 106 in Lecture Notes in Statistics, pp. 168–185. Springer-Verlag.

10. Collings, B. J. (1987). Compound random number generators. *Journal of the American Statistical Association*, Vol. 82, No. 398, pp. 525–527.

11. Compagner, A. (1991). The hierarchy of correlations in random binary sequences. *Journal of Statistical Physics*, Vol. 63, pp. 883–896.

12. Compagner, A. (1995). Operational conditions for random number generation. *Physical Review E*, Vol. 52, No. 5-B, pp. 5634–5645.

13. Couture, R. and P. L'Ecuyer (1994). On the lattice structure of certain linear congruential sequences related to AWC/SWB generators. *Mathematics of Computation*, Vol. 62, No. 206, pp. 798–808.

14. Couture, R. and P. L'Ecuyer (1995). Linear recurrences with carry as random number generators. In *Proceedings of the 1995 Winter Simulation Conference*, pp. 263–267.

15. Couture, R. and P. L'Ecuyer (1996). Computation of a shortest vector and Minkowski-reduced bases in a lattice. In preparation.

16. Couture, R. and P. L'Ecuyer (1997). Distribution properties of multiply-with-carry random number generators. *Mathematics of Computation*, Vol. 66, No. 218, pp. 591–607.

17. Couture, R., P. L'Ecuyer and S. Tezuka (1993). On the distribution of k-dimensional vectors for simple and combined Tausworthe sequences. *Mathematics of Computation*, Vol. 60, No. 202, pp. 749–761, S11–S16.

18. Coveyou, R. R. and R. D. MacPherson (1967). Fourier analysis of uniform random number generators. *Journal of the ACM*, Vol. 14, pp. 100–119.

19. Cressie, N. (1993). *Statistics for Spatial Data*. Wiley, New York.

20. De Matteis, A. and S. Pagnutti (1988). Parallelization of random number generators and long-range correlations. *Numerische Mathematik*, Vol. 53, pp. 595–608.

21. De Matteis, A. and S. Pagnutti (1990). A class of parallel random number generators.

Parallel Computing, Vol. 13, pp. 193–198.

22. Deák, I. (1990). Uniform random number generators for parallel computers. *Parallel Computing*, Vol. 15, pp. 155–164.
23. Dieter, U. (1975). How to calculate shortest vectors in a lattice. *Mathematics of Computation*, Vol. 29, No. 131, pp. 827–833.
24. Dudewicz, E. J. and T. G. Ralley (1981). *The Handbook of Random Number Generation and Testing with TESTRAND Computer Code.* American Sciences Press, Columbus, Ohio.
25. Durbin, J. (1973). *Distribution Theory for Tests Based on the Sample Distribution Function.* SIAM CBMS-NSF Regional Conference Series in Applied Mathematics. SIAM, Philadelphia.
26. Durst, M. J. (1989). Using linear congruential generators for parallel random number generation. In *Proceedings of the 1989 Winter Simulation Conference*, pp. 462–466. IEEE Press.
27. Eichenauer, J., H. Grothe, J. Lehn and A. Topuzŏglu (1987). A multiple recursive nonlinear congruential pseudorandom number generator. *Manuscripta Mathematica*, Vol. 59, pp. 331–346.
28. Eichenauer, J. and J. Lehn (1986). A nonlinear congruential pseudorandom number generator. *Statistische Hefte*, Vol. 27, pp. 315–326.
29. Eichenauer, J. and J. Lehn (1987). On the structure of quadratic congruential sequences. *Manuscripta Mathematica*, Vol. 58, pp. 129–140.
30. Eichenauer, J., J. Lehn and A. Topuzŏglu (1988). A nonlinear congruential pseudorandom number generator with power of two modulus. *Mathematics of Computation*, Vol. 51, No. 184, pp. 757–759.
31. Eichenauer-Herrmann, J. (1992). Inversive congruential pseudorandom numbers: A tutorial. *International Statistical Reviews*, Vol. 60, pp. 167–176.
32. Eichenauer-Herrmann, J. (1993). Statistical independence of a new class of inversive congruential pseudorandom numbers. *Mathematics of Computation*, Vol. 60, pp. 375–384.
33. Eichenauer-Herrmann, J. (1994). On generalized inversive congruential pseudoran-

dom numbers. *Mathematics of Computation*, Vol. 63, pp. 293–299.

34. Eichenauer-Herrmann, J. (1995). Pseudorandom number generation by nonlinear methods. *International Statistical Reviews*, Vol. 63, pp. 247–255.
35. Eichenauer-Herrmann, J. and H. Grothe (1992). A new inversive congruential pseudorandom number generator with power of two modulus. *ACM Transactions on Modeling and Computer Simulation*, Vol. 2, No. 1, pp. 1–11.
36. Eichenauer-Herrmann, J., H. Grothe and J. Lehn (1989). On the period length of pseudorandom vector sequences generated by matrix generators. *Mathematics of Computation*, Vol. 52, No. 185, pp. 145–148.
37. Eichenauer-Herrmann, J. and H. Niederreiter (1995). An improved upper bound for the discrepancy of quadratic congruential pseudorandom numbers. *Acta Arithmetica*, Vol. LXIX.2, pp. 193–198.
38. Entacher, K. (1998). Bad subsequences of well-known linear congruential pseudorandom number generators. *ACM Transactions on Modeling and Computer Simulation*, Vol. 8, No. 1. To appear.
39. Ferrenberg, A. M., D. P. Landau and Y. J. Wong (1992). Monte Carlo simulations: Hidden errors from "good" random number generators. *Physical Review Letters*, Vol. 69, No. 23, pp. 3382–3384.
40. Fincke, U. and M. Pohst (1985). Improved methods for calculating vectors of short length in a lattice, including a complexity analysis. *Mathematics of Computation*, Vol. 44, pp. 463–471.
41. Fishman, G. S. (1996). *Monte Carlo: Concepts, Algorithms, and Applications.* Springer Series in Operations Research. Springer-Verlag, New York.
42. Fishman, G. S. and L. S. Moore III (1986). An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$. *SIAM Journal on Scientific and Statistical Computing*, Vol. 7, No. 1, pp. 24–45.
43. Fushimi, M. (1983). Increasing the orders of equidistribution of the leading bits of the Tausworthe sequence. *Information Processing Letters*, Vol. 16, pp. 189–192.
44. Fushimi, M. (1989). An equivalence relation between Tausworthe and GFSR sequences and applications. *Applied Mathematics Letters*, Vol. 2, No. 2, pp. 135–137.

45. Fushimi, M. and S. Tezuka (1983). The k-distribution of generalized feedback shift register pseudorandom numbers. *Communications of the ACM*, Vol. 26, No. 7, pp. 516–523.

46. Good, I. J. (1950). *Probability and the Weighting of Evidence.* Griffin, London.

47. Good, I. J. (1969). How random are random numbers? *The American Statistician*, Vol. , pp. 42–45.

48. Grothe, H. (1987). Matrix generators for pseudo-random vectors generation. *Statistische Hefte*, Vol. 28, pp. 233–238.

49. Hellekalek, P. (1995). Inversive pseudorandom number generators: Concepts, results, and links. In C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman, editors, *Proceedings of the 1995 Winter Simulation Conference*, pp. 255–262. IEEE Press.

50. Hoaglin, D. C. and M. L. King (1978). A remark on algorithm AS 98: The spectral test for the evaluation of congruential pseudo-random generators. *Applied Statistics*, Vol. 27, pp. 375–377.

51. Hörmann, W. and G. Derflinger (1993). A portable random number generator well suited for the rejection method. *ACM Transactions on Mathematical Software*, Vol. 19, No. 4, pp. 489–495.

52. Huber, K. (1994). On the period length of generalized inversive pseudorandom number generators. *Applied Algebra in Engineering, Communications, and Computing*, Vol. 5, pp. 255–260.

53. Hull, T. E. (1962). Random number generators. *SIAM Review*, Vol. 4, pp. 230–254.

54. IMSL (1987). *IMSL Library Users's Manual, Vol.3.* IMSL, Houston, Texas.

55. James, F. (1990). A review of pseudorandom number generators. *Computer Physics Communications*, Vol. 60, pp. 329–344.

56. James, F. (1994). RANLUX: A Fortran implementation of the high-quality pseudorandom number generator of Lüscher. *Computer Physics Communications*, Vol. 79, pp. 111–114.

57. Knuth, D. E. (1981). *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, second edition. Addison-Wesley, Reading, Mass.

58. Koç, C. (1995). Recurring-with-carry sequences. *Journal of Applied Probability*, Vol.

32, pp. 966–971.

59. Lagarias, J. C. (1993). Pseudorandom numbers. *Statistical Science*, Vol. 8, No. 1, pp. 31–39.

60. Law, A. M. and W. D. Kelton (1991). *Simulation Modeling and Analysis*, second edition. McGraw-Hill, New York.

61. L'Ecuyer, P. (1988). Efficient and portable combined random number generators. *Communications of the ACM*, Vol. 31, No. 6, pp. 742–749 and 774. See also the correspondence in the same journal, Vol. 32, No. 8 (1989), pp. 1019–1024.

62. L'Ecuyer, P. (1990). Random numbers for simulation. *Communications of the ACM*, Vol. 33, No. 10, pp. 85–97.

63. L'Ecuyer, P. (1992). Testing random number generators. In *Proceedings of the 1992 Winter Simulation Conference*, pp. 305–313. IEEE Press.

64. L'Ecuyer, P. (1994). Uniform random number generation. *Annals of Operations Research*, Vol. 53, pp. 77–120.

65. L'Ecuyer, P. (1996). Combined multiple recursive random number generators. *Operations Research*, Vol. 44, No. 5, pp. 816–822.

66. L'Ecuyer, P. (1996). Maximally equidistributed combined Tausworthe generators. *Mathematics of Computation*, Vol. 65, No. 213, pp. 203–213.

67. L'Ecuyer, P. (1997). Bad lattice structures for vectors of non-successive values produced by some linear recurrences. *INFORMS Journal on Computing*, Vol. 9, No. 1, pp. 57–60.

68. L'Ecuyer, P. (1997). Good parameters and implementations for combined multiple recursive random number generators. Manuscript.

69. L'Ecuyer, P. (1997). Tests based on sum-functions of spacings for uniform random numbers. *Journal of Statistical Computation and Simulation*, Vol. 59, pp. 251–269.

70. L'Ecuyer, P. (1998). Tables of maximally equidistributed combined LFSR generators. *Mathematics of Computation*, To appear.

71. L'Ecuyer, P. (Circa 2000). TestU01: Un logiciel pour appliquer des tests statistiques à des générateurs de valeurs aléatoires. In preparation.

72. L'Ecuyer, P. and T. H. Andres (1997). A random number generator based on the

combination of four LCGs. *Mathematics and Computers in Simulation*, Vol. 44, pp. 99–107.

73. L'Ecuyer, P., F. Blouin and R. Couture (1993). A search for good multiple recursive random number generators. *ACM Transactions on Modeling and Computer Simulation*, Vol. 3, No. 2, pp. 87–98.
74. L'Ecuyer, P., A. Compagner and J.-F. Cordeau (1997). Entropy tests for random number generators. Manuscript.
75. L'Ecuyer, P., J.-F. Cordeau and R. Simard (1997). Close-point spatial tests and their application to random number generators. Submitted.
76. L'Ecuyer, P. and S. Côté (1991). Implementing a random number package with splitting facilities. *ACM Transactions on Mathematical Software*, Vol. 17, No. 1, pp. 98–111.
77. L'Ecuyer, P. and R. Couture (1997). An implementation of the lattice and spectral tests for multiple recursive linear random number generators. *INFORMS Journal on Computing*, Vol. 9, No. 2, pp. 206–217.
78. L'Ecuyer, P. and R. Proulx (1989). About polynomial-time "unpredictable" generators. In *Proceedings of the 1989 Winter Simulation Conference*, pp. 467–476. IEEE Press.
79. L'Ecuyer, P., R. Simard and S. Wegenkittl (1998). Sparse serial tests of randomness. In preparation.
80. L'Ecuyer, P. and S. Tezuka (1991). Structural properties for two classes of combined random number generators. *Mathematics of Computation*, Vol. 57, No. 196, pp. 735–746.
81. Leeb, H. and S. Wegenkittl (1997). Inversive and linear congruential pseudorandom number generators in empirical tests. *ACM Transactions on Modeling and Computer Simulation*, Vol. 7, No. 2, pp. 272–286.
82. Lehmer, D. H. (1951). Mathematical methods in large scale computing units. *Annals Comp. Laboratory Harvard University*, Vol. 26, pp. 141–146.
83. Lewis, P. A. W., A. S. Goodman and J. M. Miller (1969). A pseudo-random number generator for the system/360. *IBM System's Journal*, Vol. 8, pp. 136–143.

84. Lewis, T. G. and W. H. Payne (1973). Generalized feedback shift register pseudorandom number algorithm. *Journal of the ACM*, Vol. 20, No. 3, pp. 456–468.

85. Lüscher, M. (1994). A portable high-quality random number generator for lattice field theory simulations. *Computer Physics Communications*, Vol. 79, pp. 100–110.

86. MacLaren, N. M. (1992). A limit on the usable length of a pseudorandom sequence. *Journal of Statistical Computing and Simulation*, Vol. 42, pp. 47–54.

87. Marsaglia, G. (1985). A current view of random number generators. In *Computer Science and Statistics, Sixteenth Symposium on the Interface*, pp. 3–10, North-Holland, Amsterdam. Elsevier Science Publishers.

88. Marsaglia, G. (1994). Yet another rng. Posted to the electronic billboard `sci.stat.math`, August 1.

89. Marsaglia, G. (1996). DIEHARD: a battery of tests of randomness. See `http://stat.fsu.edu/~geo/diehard.html`.

90. Marsaglia, G. (1996). The Marsaglia random number CDROM. See `http://stat.fsu.edu/~geo/`.

91. Marsaglia, G. and A. Zaman (1991). A new class of random number generators. *The Annals of Applied Probability*, Vol. 1, pp. 462–480.

92. Marse, K. and S. D. Roberts (1983). Implementing a portable FORTRAN uniform (0,1) generator. *Simulation*, Vol. 41, No. 4, pp. 135–139.

93. Mascagni, M., M. L. Robinson, D. V. Pryor and S. A. Cuccaro (1995). Parallel pseudorandom number generation using additive lagged-fibonacci recursions. In H. Niederreiter and P. J.-S. Shiue, editors, *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, number 106 in Lecture Notes in Statistics, pp. 263–277. Springer-Verlag.

94. MATLAB (1992). *MATLAB Reference Manual*. The MathWorks Inc., Natick, Mass.

95. Matsumoto, M. and Y. Kurita (1992). Twisted GFSR generators. *ACM Transactions on Modeling and Computer Simulation*, Vol. 2, No. 3, pp. 179–194.

96. Matsumoto, M. and Y. Kurita (1994). Twisted GFSR generators II. *ACM Transactions on Modeling and Computer Simulation*, Vol. 4, No. 3, pp. 254–266.

97. Matsumoto, M. and Y. Kurita (1996). Strong deviations from randomness in m-

sequences based on trinomials. *ACM Transactions on Modeling and Computer Simulation*, Vol. 6, No. 2, pp. 99–106.

98. Matsumoto, M. and T. Nishimura (1998). Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, Vol. 8, No. 1. To appear.

99. Moreau, T. (1996). A practical "perfect" pseudo-random number generator. Manuscript.

100. Niederreiter, H. (1985). The serial test for pseudorandom numbers generated by the linear congruential method. *Numerische Mathematik*, Vol. 46, pp. 51–68.

101. Niederreiter, H. (1986). A pseudorandom vector generator based on finite field arithmetic. *Mathematica Japonica*, Vol. 31, pp. 759–774.

102. Niederreiter, H. (1992). *Random Number Generation and Quasi-Monte Carlo Methods*, volume 63 of *SIAM CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM, Philadelphia.

103. Niederreiter, H. (1995). The multiple-recursive matrix method for pseudorandom number generation. *Finite Fields and their Applications*, Vol. 1, pp. 3–30.

104. Niederreiter, H. (1995). New developments in uniform pseudorandom number and vector generation. In H. Niederreiter and P. J.-S. Shiue, editors, *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, number 106 in Lecture Notes in Statistics, pp. 87–120. Springer-Verlag.

105. Niederreiter, H. (1995). Pseudorandom vector generation by the multiple-recursive matrix method. *Mathematics of Computation*, Vol. 64, No. 209, pp. 279–294.

106. Owen, A. B. (1998). Latin supercube sampling for very high dimensional simulations. *ACM Transactions of Modeling and Computer Simulation*, Vol. 8, No. 1. To appear.

107. Park, S. K. and K. W. Miller (1988). Random number generators: Good ones are hard to find. *Communications of the ACM*, Vol. 31, No. 10, pp. 1192–1201.

108. Payne, W. H., J. R. Rabung and T. P. Bogyo (1969). Coding the Lehmer pseudorandom number generator. *Communications of the ACM*, Vol. 12, pp. 85–86.

109. Percus, D. E. and M. Kalos (1989). Random number generators for MIMD parallel

processors. *Journal of Parallel and Distributed Computation*, Vol. 6, pp. 477–497.

110. Press, W. H. and S. A. Teukolsky (1992). Portable random number generators. *Computers in Physics*, Vol. 6, No. 5, pp. 522–524.

111. Rabin, M. O. (1980). Probabilistic algorithms for primality testing. *J. Number Theory*, Vol. 12, pp. 128–138.

112. Ripley, B. D. (1987). *Stochastic Simulation*. Wiley, New York.

113. Ripley, B. D. (1990). Thoughts on pseudorandom number generators. *Journal of Computational and Applied Mathematics*, Vol. 31, pp. 153–163.

114. Schrage, L. (1979). A more portable fortran random number generator. *ACM Transactions on Mathematical Software*, Vol. 5, pp. 132–138.

115. Stephens, M. S. (1986). Tests based on EDF statistics. In R. B. D'Agostino and M. S. Stephens, editors, *Goodness-of-Fit Techniques*. Marcel Dekker, New York and Basel.

116. Stephens, M. S. (1986). Tests for the uniform distribution. In R. B. D'Agostino and M. S. Stephens, editors, *Goodness-of-Fit Techniques*, pp. 331–366. Marcel Dekker, New York and Basel.

117. Sun Microsystems (1991). *Numerical Computations Guide*. Document number 800-5277-10.

118. Tausworthe, R. C. (1965). Random numbers generated by linear recurrence modulo two. *Mathematics of Computation*, Vol. 19, pp. 201–209.

119. Teichroew, D. (1965). A history of distribution sampling prior to the era of computer and its relevance to simulation. *Journal of the American Statistical Association*, Vol. 60, pp. 27–49.

120. Tezuka, S. (1995). *Uniform Random Numbers: Theory and Practice*. Kluwer Academic Publishers, Norwell, Mass.

121. Tezuka, S. and P. L'Ecuyer (1991). Efficient and portable combined Tausworthe random number generators. *ACM Transactions on Modeling and Computer Simulation*, Vol. 1, No. 2, pp. 99–112.

122. Tezuka, S., P. L'Ecuyer and R. Couture (1994). On the add-with-carry and subtract-with-borrow random number generators. *ACM Transactions of Modeling*

and Computer Simulation, Vol. 3, No. 4, pp. 315–331.

123. Tootill, J. P. R., W. D. Robinson and D. J. Eagle (1973). An asymptotically random Tausworthe sequence. *Journal of the ACM*, Vol. 20, pp. 469–481.

124. Vazirani, U. and V. Vazirani (1984). Efficient and secure pseudo-random number generation. In *Proceedings of the 25th IEEE Symposium on Foundations of Computer Science*, pp. 458–463.

125. Wang, D. and A. Compagner (1993). On the use of reducible polynomials as random number generators. *Mathematics of Computation*, Vol. 60, pp. 363–374.

126. Wichmann, B. A. and I. D. Hill (1982). An efficient and portable pseudo-random number generator. *Applied Statistics*, Vol. 31, pp. 188–190. See also corrections and remarks in the same journal by Wichmann and Hill, Vol. 33 (1984) p. 123; McLeod Vol. 34 (1985) pp. 198–200; Zeisel Vol. 35 (1986) p. 89.

127. Wolfram, S. (1996). *The Mathematica Book*, third edition. Wolfram Media/Cambridge University Press, Champaign, USA.