PSEUDORANDOM NUMBER GENERATORS

Pierre L'Ecuyer

DIRO, Université de Montreal
C.P. 6128, Succ. Centre-Ville
Montréal (Québec), Canada, H3C 3J7
http://www.iro.umontreal.ca/~lecuyer/

August 28, 2007

KEYWORDS

Uniform random numbers, multiple streams, common random numbers, linear congruential

ABSTRACT

We review the basic principles underlying the design of uniform random number generators, their main quality requirements, their theoretical analysis, and their empirical testing. The main construction techniques of algorithmic generators are discussed, with an emphasis on the most common ones, the linear generators. Nonlinear generators, as well as hardware-based generators, are also briefly discussed. For simulation, we explain why we favor generators that can offer multiple streams and substreams, and we recommend specific ones.

INTRODUCTION

Stochastic models of quantitative finance are defined in the abstract framework of probability theory. To apply the Monte Carlo method to these models, it suffices in principle to sample independent realizations of the underlying random variables or random vectors. This can be achieved by sampling independent random variables uniformly distributed over the interval $(0,1)$ (i.i.d. $\mathscr{U}(0,1)$, for short) and applying appropriate transformations to these uniform random variables. Non-uniform variate generation techniques develop such transformations and provide efficient algorithms that implement them [3, 6]. A simple general way to obtain independent random variables $X_1, X_2, \ldots$ with distribution function F from a sequence of i.i.d. $\mathscr{U}(0,1)$ random variables $U_1, U_2, \ldots$ is to define

$$X_j = F^{-1}(U_j) \stackrel{\text{def}}{=} \min\{x \mid F(x) \geq U_j\}; \tag{1}$$

this is the *inversion method*. This technique can provide a sequence of independent standard normal random variables, for example, which can in turn be used to generate the sample path of a geometric Brownian motion or other similar type of stochastic process. There is no closed form expression for the inverse standard normal distribution function, but very accurate numerical approximations are available.

But how do we get the i.i.d. $\mathscr{U}(0,1)$ random variables? Realizing these random variables exactly is very difficult, perhaps practically impossible. With current knowledge, this can be realized only *approximately*. Fortunately, the approximation seems good enough for all practical applications of the Monte Carlo method in financial engineering and in other areas as well.

A first class of methods to realize approximations of these random variables are based on real physical noise coming from hardware devices. There is a large variety of such devices; they include gamma ray counters, fast oscillators sampled at low and slightly random frequencies, amplifiers of heat noise produced in electric resistances, photon counting and photon trajectory detectors, and so on. Some of these devices sample a signal at successive epochs and return 0 if the signal is below a given threshold, and 1 if it is above the threshold, at each sampling epoch. Others return the parity of a counter. Most of them produce sequences of bits that are slightly correlated and often slightly biased, but the bias and correlation can be reduced to a negligible amount, that becomes practically undetectable by statistical tests in reasonable time, by combining the bits in a clever

way. For example, a simple technique to eliminate the bias when there is no correlation, proposed long ago by John von Neumann, places the successive bits in non-overlapping pairs, discards all the pairs 00 and 11, and replaces the pairs 01 and 10 by 1 and 0, respectively. Generalizations of this technique can eliminate both the bias and correlation [2]. Simpler techniques such as xoring (adding modulo 2) the bits by blocks of 2 or more, or xoring several bit streams from different sources, are often used in practice. Reliable devices to generate random bits and numbers, based on these techniques, are available on the market. These types of devices are needed for applications such as cryptography, lotteries, and gambling machines, for example, where some amount of real randomness (or entropy) is essential to provide the required unpredictability and security.

For Monte Carlo methods, however, these devices are unnecessary and unpractical. They are unnecessary because simple *deterministic algorithms* are available that require no other hardware than a standard computer and provide good enough imitations of i.i.d. $U(0,1)$ random variables from a statistical viewpoint, in the sense that the statistical behavior of the simulation output is pretty much the same (for all practical purposes) if we use *(pseudo)random numbers* produced by these algorithms in place of true i.i.d. $U(0,1)$ random variables. These deterministic algorithmic methods are much more convenient than hardware devices.

A *(pseudo)random number generator* (RNG, for short) can be defined as a structure comprised of the following ingredients [9]: a finite *set of states* $\mathcal{S}$; a probability distribution on $\mathcal{S}$ to select the *initial state* s_0 (also called the *seed*); a *transition function* $f : \mathcal{S} \to \mathcal{S}$; an *output space* $\mathcal{U}$; and an *output function* $g : \mathcal{S} \to \mathcal{U}$. Here we assume that $\mathcal{U}$ is the interval $(0,1)$. The state evolves according to the recurrence $s_i = f(s_{i-1})$, for $i \geq 1$, and the *output* at step i is $u_i = g(s_i) \in \mathcal{U}$. These u_i's are the successive *random numbers* produced by the RNG. (Following common usage in the simulation community, here we leave out the qualifier "pseudo." In the area of cryptology, the term *pseudorandom number generator* refers to a stronger notion, with polynomial-time unpredictability properties [20]).

Because $\mathcal{S}$ is finite, the RNG must eventually return to a previously visited state, i.e., $s_{l+j} = s_l$ for some $l \geq 0$ and $j > 0$. Then, $s_{i+j} = s_i$ and $u_{i+j} = u_i$ for all $i \geq l$; i.e., the output sequence eventually repeats itself. The smallest $j > 0$ for which this happens is the *period length* ρ. Clearly, ρ cannot exceed $|\mathcal{S}|$, the number of distinct states. If the state can be represented with b bits of memory, then $\rho \leq 2^b$. For good RNGs, ρ is usually close to 2^b, as it is not difficult to construct

recurrences with this property. Typical values of b range from 31 to around 20,000 or even higher [18]. In our opinion, ρ should never be less than 2^{100}, and preferably more than 2^{200}. Values of b that exceed 1000 are unnecessary if the RNG satisfies the quality criteria described in what follows.

A key advantage of algorithmic RNGs is their ability to repeat exactly the same sequence of random numbers without storing them. Repeating the same sequence several times is essential for the proper implementation of variance reduction techniques such as using common random numbers for comparing similar systems, for sensitivity analysis, for sample-path optimization, for external control variates, for antithetic variates, and so on [1, 5] (see also eqf13-021, eqf13-022). It is also handy for program verification and debugging. On the other hand, some real randomness can be used for selecting the seed s_0 of the RNG.

STREAMS AND SUBSTREAMS

Modern high-quality simulation software often offers the possibility to declare and create virtual RNGs just like for any other type of variable or object, in practically unlimited amount. In an implementation adopted by several simulation software vendors, these virtual RNGs are called *streams*, and each stream is split into multiple *substreams* long enough to prevent potential overlap [19, 14]. For any given stream, there are methods to generate the next number, to rewind to the beginning of the stream, or to the beginning of the current substream, or to the beginning of the next substream.

To illustrate why this is useful, consider a simple model of a financial option whose payoff is a function of a geometric Brownian motion observed at fixed points in time. We want to estimate $d = \mathbb{E}[X_2 - X_1]$ where X_1 and X_2 are the payoffs with two slightly different parameter settings, such as different volatilities or different strike prices, for example. This is often useful for sensitivity analysis (estimating the *greeks*; see eqf13-004). To estimate d we would simulate the model with the two different settings using *common random numbers* across the two versions [1, 5] (see also eqf13-021), repeat this n times independently, and compute a confidence interval on d from the n independent copies of $X_2 - X_1$. To implement this, we take a stream of random numbers that contains multiple substreams, use the same substream to simulate both X_1 and X_2 for each replication,

and n different substreams for the n replications. At the beginning of a replication, the stream is placed to the beginning of a new substream and the model is simulated to compute X_1. Then the stream is reset to the beginning of its current substream before simulating the model again to compute X_2. This ensures that exactly the same random numbers are used to generate the Brownian motion increments at the same time points for both X_1 and X_2. Then the stream is moved to the beginning of the next substream for the next pair of runs.

There are many situations where the number of calls to the RNG during a simulation depends on the model parameters, and may not be the same for X_1 and X_2. Even in that case, the above scheme ensures that the RNG restarts at the same place for both parameter settings, for each replication. In more complicated models, to ensure a good synchronization of the random numbers across the two settings (i.e., make sure that the same random numbers are used for the same purposes in both cases), it is typically convenient to have several different streams, each stream being dedicated to one specific aspect of the model. For instance, in the previous example, if we also need to simulate external events that occur according to a Poisson process and influence the payoff in some way (e.g., they could trigger jumps in the Brownian motion), it is better to use a separate stream to simulate this process, to guarantee that no random number is used for the Brownian motion increment in one setting and for the Poisson process in the other setting.

QUALITY CRITERIA AND TESTING

A good RNG must obviously have a very long period, to make sure that there is no chance of wrapping around. It should also be *repeatable* (able to reproduce exactly the same sequence several times), *portable* (be easy to implement and behave the same way in different software/hardware environments), and it should be easy to split its sequence into several disjoint streams and substreams, and implement efficient tools to move between those streams and substreams. The latter requires the availability of efficient jump-ahead methods, that can quickly compute $s_{i+\nu}$ given s_i, for any large ν. The number b of bits required to store the state should not be too large, because the cost (CPU time) of jumping-ahead typically increases faster than linearly with b, and also because there can be a large number of streams and substreams in a given simulation, especially for large complex models. Another key performance measure is the speed of the generator itself. Fast

generators can produce up to 100 million $\mathscr{U}(0,1)$ random numbers per second on current personal computers [18].

All these nice properties are not sufficient, however. For example, an RNG that returns $u_i = (i/10^{1000}) \mod 1$ at step i satisfies these properties but is definitely not recommendable, because its successive output values have an obvious strong correlation. Ideally, if we select a random seed s_0 uniformly in $\mathscr{S}$, we would like the vector of the first s output values, $(u_0, \ldots, u_{s-1})$, to be uniformly distributed over the s-dimensional unit hypercube $[0,1]^s$ for each $s > 0$. This would guarantee both uniformity and independence. Formally, we cannot have this, because these s-dimensional vectors must take their values from the finite set $\Psi_s = \{(u_0, \ldots, u_{s-1}) : s_0 \in \mathscr{S}\}$, whose cardinality cannot exceed $|\mathscr{S}|$. If s_0 is random, Ψ_s can be viewed as the *sample space* from which vectors of successive output values are drawn randomly. Then, to approximate the uniformity and independence, we want the finite set Ψ_s to provide a dense and uniform coverage of the hypercube $[0,1]^s$, at least for small and moderate values of s. This is possible only if $\mathscr{S}$ has large cardinality, and it is in fact a more important reason for having a long period than the danger of exhausting the cycle.

So the uniformity of Ψ_s in $[0,1]^s$ is a key quality criterion. But how do we measure it? There are many ways of measuring the uniformity (or the discrepancy from the uniform distribution) for a point set in the unit hypercube [16, 22] (see also eqf13-019). To be practical, the uniformity measure must be selected so that it can be effectively computed without generating explicitly the points of Ψ_s. For this reason, the theoretical *figures of merit* that measure the uniformity usually depend on the mathematical structure of the RNG. This is also the main reason for RNGs based on linear recurrences: their point sets Ψ_s are easier to analyze mathematically, because they have a simpler structure. One could argue that nonlinear and more complex structures give rise to point sets Ψ_s that look more random, and some of them behave very well in empirical statistical tests, but their structure is much harder to analyze. They could leave large holes in $[0,1]^s$ that are difficult to detect.

To design a good RNG, one typically selects an algorithm together with the size of the state space, and constraints on the parameters that ensure a fast implementation. Then one makes a computerized search in the space of parameters to find a set of values that give (a) the maximal period length within this class of generators and then (b) the largest figure of merit than can be

found. RNGs are thus selected and constructed based primarily on theoretical criteria. Then, they are implemented and tested empirically.

A large variety of *empirical statistical tests* have been designed and implemented for RNGs [8, 18]. All these tests try to detect empirical evidence against the hypothesis $\mathscr{H}_0$ that the u_i are i.i.d. $\mathscr{U}[0,1]$. A test can be any function Y of a finite set of u_i's, that can be computed in reasonable time, and whose distribution under $\mathscr{H}_0$ can be approximated well enough. There is an unlimited number of such tests. When applying the test, one computes the realization of Y, say y, and then the probability $p^+ = \mathbb{P}[Y \geq y \mid \mathscr{H}_0]$, called the right p-value. If Y takes a much larger value than expected, then p^+ will be very close to zero, and we declare that the RNG fails the test. We may also examine the left p-value $p^- = \mathbb{P}[Y \leq y \mid \mathscr{H}_0]$, or both p^+ and p^-, depending on the design of the test. When a generator really fails a test, it is not unusual to find p-values as small as 10^{-15} or less.

Specific batteries that contain a variety of standard tests, that detect problems often encountered in poorly designed or too simple RNGs, have been proposed and implemented [18]. The bad news is that a majority of the RNGs available in popular commercial software fail these tests unequivocally, with p-values smaller than 10^{-15}. These generators should be discarded, unless we have very good reasons to believe that for our specific simulation models, the problems detected by these failed tests will not affect the results. The good news is that some freely-available high-quality generators pass all the tests in these batteries. Of course, passing all these tests is not a proof that the RNG is reliable for all possible simulations; but it certainly improves our confidence in the generator. In fact, no RNG can pass all conceivable statistical tests. In some sense, the good RNGs fail only very complicated tests that are hard to find and implement, whereas bad RNGs fail simple tests.

LINEAR RECURRENCES

Most RNGs used for simulation are based on linear recurrences of the general form

$$x_i \;=\; (a_1 x_{i-1} + \cdots + a_k x_{i-k}) \mod m, \tag{2}$$

where k and m are positive integers, and the coefficients $a_1, \ldots, a_k$ are in $\{0, 1, \ldots, m-1\}$, with $a_k \neq 0$. Some use a large value of m, preferably a prime number, and define the output as $u_i = x_i/m$, so the state at step i can be viewed as $s_i = \mathbf{x}_i = (x_{i-k+1}, \ldots, x_i)$. The RNG is then called a *multiple recursive generator* (MRG). For $k = 1$, we obtain the classical *linear congruential generator* (LCG). In practice, the output transformation is modified slightly to make sure that u_i is always strictly between 0 and 1, for example by taking $u_i = (x_i + 1)/(m+1)$ or $u_i = (x_i + 1/2)/m$. Jumping ahead from $\mathbf{x}_i$ to $\mathbf{x}_{i+\nu}$ for an arbitrary large ν can be implemented easily: because of the linearity, one can write $\mathbf{x}_{i+\nu} = \mathbf{A}_\nu \mathbf{x}_i \mod m$, where $\mathbf{A}_\nu$ is a $k \times k$ matrix that can be precomputed once for all [13]. When m is prime, one can choose the coefficients a_j so that the period length reaches $m^k - 1$, its maximum [8].

The point set Ψ_s produced by an MRG is known to have a *lattice structure*, and its uniformity is measured via a figure of merit for the quality of that lattice, for several values of s. This is known as the *spectral test* [8, 4, 10].

Typically, m is chosen as one of the largest prime integers representable on the target computer, e.g., $m = 2^{31} - 1$ on a 32-bit computer. Then, a direct implementation of (2) with integer numbers would cause overflow, so more clever implementation techniques are needed. These techniques require that we impose additional conditions on the coefficients a_j. We have to be careful that these conditions do not oversimplify the structure of the point set Ψ_s. One extreme example of this is to take only two nonzero coefficients, say a_r and a_k, both equal to ± 1. Implementation is then easy and fast. However, all triples of the form (u_i, u_{i-r}, u_{i-k}) produced by such a generator, for $i = 0, 1, \ldots$, lie in only two planes in the three-dimensional unit cube! Despite this awful behavior, these types of generators (or variants thereof) can be found in many popular software products [18]. They should be avoided. All simple LCGs, say with $m \leq 2^{64}$, should be discarded; they have too much structure and their period length is too short for today's computers.

One effective way of implementing high-quality MRGs is to combine two (or more) of them by adding their outputs modulo 1. (There are also other slightly different ways of combining.) If the components have distinct prime moduli, the combination turns out to be just another MRG with (non-prime) modulus m equal to the product of the moduli of the components, and the period can be up to half the product of the component's periods when we combine two of them. The idea is to select the components so that (a) a fast implementation is easy to construct for each individual

component, and (b) the combined MRG has a more complicated structure and highly-uniform sets Ψ_s, as measured by the spectral test [10]. Specific MRG constructions can be found in [10, 13, 18] and the references given there.

A different approach uses a linear recurrence as in (2), but with $m = 2$. All operations are then performed modulo 2, i.e., in the finite field $\mathbb{F}_2$ with elements $\{0,1\}$. This allows very fast implementations by exploiting the binary nature of computers. A general framework for this is the matrix linear recurrence [13, 17]:

$$\mathbf{x}_i = \mathbf{A}\mathbf{x}_{i-1}, \tag{3}$$

$$\mathbf{y}_i = \mathbf{B}\mathbf{x}_i, \tag{4}$$

$$u_i = \sum_{\ell=1}^{w} y_{i,\ell-1} 2^{-\ell} \tag{5}$$

where $\mathbf{x}_i = (x_{i,0}, \ldots, x_{i,k-1})^{\mathsf{t}}$ is the k-bit *state vector* at step i, $\mathbf{y}_i = (y_{i,0}, \ldots, y_{i,w-1})^{\mathsf{t}}$ is the w-bit *output vector* at step i, k and w are positive integers, $\mathbf{A}$ is a $k \times k$ binary *transition matrix*, $\mathbf{B}$ is a $w \times k$ binary *output transformation matrix*, and $u_i \in [0,1)$ is the *output* at step i. All operations in (3) and (4) are performed in $\mathbb{F}_2$. These RNGs are called $\mathbb{F}_2$-linear generators.

The theoretical analysis usually assumes the simple output definition (5), but in practice this definition is modified slightly to avoid returning 0 or 1. This framework covers several types of generators, including the Tausworthe, polynomial LCG, generalized feedback shift register (GFSR), twisted GFSR, Mersenne twister, WELL, xorshift, linear cellular automaton, and combinations of these [13, 17, 21]. With a carefully selected matrix $\mathbf{A}$ (its characteristic polynomial must be a primitive polynomial over $\mathbb{F}_2$), the period length can reach $2^k - 1$. In practice, the matrices $\mathbf{A}$ and $\mathbf{B}$ are chosen so that the products (3) and (4) can be implemented very efficiently on a computer by a few simple binary operations such as or, exclusive-or, shift, and rotation, on blocks of bits. The idea is to find a compromise between the number of such operations (which affects the speed) and a good uniformity of the point sets Ψ_s (which is easier to reach with more operations). The uniformity of these point sets is measured via their *equidistribution*; essentially, the hypercube $[0,1]^s$ is partitioned into small subcubes (or subrectangles) of equal sizes, and for several such partitions, we check if all the subcubes contain exactly the same number of points from Ψ_s. This can be computed efficiently by computing the ranks of certain binary matrices [17]. Combined generators of this type, defined by xoring the output vectors $\mathbf{y}_i$ of the components, are equivalent to yet another $\mathbb{F}_2$-linear generator. Such combinations have the same motivation as for MRGs.

NONLINEAR GENERATORS

Linear RNGs have many nice properties, but they also fail certain specialized statistical tests focused at detecting linearity. When the simulation itself applies nonlinear transformations to the uniform random numbers, which is typical, one should not worry about the linearity, unless the structure of Ψ_s is not very good. But there are cases where the linearity can matter. For example, to generate a large random binary matrix, one should not use an $\mathbb{F}_2$-linear generator, because the rank of the matrix is likely to be much smaller than expected, due to the excessive linear dependence [18].

There are many ways of constructing nonlinear generators. For example, one can simply add a nonlinear output transformation to a linear RNG, or permute (shuffle) the output values with the help of another generator. Another way is to combine an MRG with and $\mathbb{F}_2$-linear generator, either by addition modulo 1 or by xoring the outputs. An important advantage of this technique is that the uniformity of the resulting combined generator can be assessed theoretically, at least to a certain extent [15]. They can also be fast.

When combining generators, it is important to understand what we do and we should be careful to examine the structure of the combination, not only of the quality of the components. By blindly combining two good components, it is indeed possible (and not too difficult) to obtain a bad (worst) RNG.

Generators whose underlying recurrence is nonlinear are generally harder to analyze and are slower. These are the types of generators used for cryptographic applications. Empirically, well-designed nonlinear generators tend to perform better in statistical tests than the linear ones [18], but from the theoretical perspective, their structure is not understood as well. RNGs based on chaotic dynamical systems have often been proposed in the literature, but these generators have several major drawbacks, including the fact that their s-dimensional uniformity is often very poor [7].

WHAT TO LOOK FOR AND WHAT TO AVOID

A quick look at the empirical results in [12, 18] shows that many widely-used RNGs are seriously deficient, including the default generators of several highly popular software products. So before running important simulation experiments, one should always check what is the default RNG, and be ready to replace it if needed. Note that the generators that pass the tests in [18] are not all recommended. Before adoption, one should verify that the RNG has solid theoretical support, that it is fast enough, and that multiple streams and substreams are available, for example. Convenient software packages with multiple streams and substreams are described in [19, 14] and are available freely from the web page of this author. These packages are based on combined MRGs of [10], combined Tausworthe generators of [11], the WELL generators [23] (which are improvements over the Mersenne twister in terms of equidistribution), and some additional nonlinear generators, among others. No uniform RNG can be guaranteed against all possible defects, but one should at least avoid those that fail simple statistical tests miserably and go for the more robust ones, for which no serious problem has been detected after years of usage and testing.

ACKNOWLEDGEMENTS

This work has been supported by the Natural Sciences and Engineering Research Council of Canada Grant No. ODGP0110050 and a Canada Research Chair to the author.

REFERENCES

1. P. Bratley, B. L. Fox, and L. E. Schrage. *A Guide to Simulation*. Springer-Verlag, New York, second edition, 1987.
2. B. Chor and O. Goldreich. Unbiased bits from sources of weak randomness and probabilistic communication complexity. *SIAM Journal on Computation*, 17(2):230–261, 1988.
3. L. Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, 1986.
4. G. S. Fishman. *Monte Carlo: Concepts, Algorithms, and Applications*. Springer Series in Operations Research. Springer-Verlag, New York, 1996.

5. P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer-Verlag, New York, 2004.
6. W. Hörmann, J. Leydold, and G. Derflinger. *Automatic Nonuniform Random Variate Generation*. Springer-Verlag, Berlin, 2004.
7. P. Jäckel. *Monte Carlo Methods in Finance*. John Wiley, Chichester, U.K., 2002.
8. D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, Mass., third edition, 1998.
9. P. L'Ecuyer. Uniform random number generation. *Annals of Operations Research*, 53:77–120, 1994.
10. P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, 1999.
11. P. L'Ecuyer. Tables of maximally equidistributed combined LFSR generators. *Mathematics of Computation*, 68(225):261–269, 1999.
12. P. L'Ecuyer. Software for uniform random number generation: Distinguishing the good and the bad. In *Proceedings of the 2001 Winter Simulation Conference*, pages 95–105, Pistacaway, NJ, 2001. IEEE Press.
13. P. L'Ecuyer. Uniform random number generation. In S. G. Henderson and B. L. Nelson, editors, *Simulation*, Handbooks in Operations Research and Management Science, pages 55–81. Elsevier, Amsterdam, The Netherlands, 2006. Chapter 3.
14. P. L'Ecuyer and E. Buist. Simulation in Java with SSJ. In *Proceedings of the 2005 Winter Simulation Conference*, pages 611–620. IEEE Press, 2005.
15. P. L'Ecuyer and J. Granger-Piché. Combined generators with components from different families. *Mathematics and Computers in Simulation*, 62:395–404, 2003.
16. P. L'Ecuyer and C. Lemieux. Recent advances in randomized quasi-Monte Carlo methods. In M. Dror, P. L'Ecuyer, and F. Szidarovszky, editors, *Modeling Uncertainty: An Examination of Stochastic Theory, Methods, and Applications*, pages 419–474. Kluwer Academic, Boston, 2002.
17. P. L'Ecuyer and F. Panneton. $\mathbf{F}_2$-linear random number generators. Submitted for publication, 2007.
18. P. L'Ecuyer and R. Simard. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4):Article 22, August 2007.
19. P. L'Ecuyer, R. Simard, E. J. Chen, and W. D. Kelton. An object-oriented random-number

package with many long streams and substreams. *Operations Research*, 50(6):1073–1075, 2002.

20. M. Luby. *Pseudorandomness and Cryptographic Applications*. Princeton University Press, Princeton, 1996.
21. M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.
22. H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*, volume 63 of *SIAM CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM, Philadelphia, 1992.
23. F. Panneton, P. L'Ecuyer, and M. Matsumoto. Improved long-period generators based on linear recurrences modulo 2. *ACM Transactions on Mathematical Software*, 32(1):1–16, 2006.