Draft revised entry for International Encyclopedia of Statistical Science, second edition, Lovric, Miodrag (Ed.), Springer-Verlag, 2024.

Uniform Random Number Generators

Pierre L'Ecuyer
Professor, DIRO
Université de Montréal, Montréal, QC, Canada
http://www.iro.umontreal.ca/~lecuyer/

Introduction

▶Monte Carlo methods are at the core of modern computational statistics (Robert and Casella, 2004; Owen, 2023); see also ▶Monte Carlo methods in statistics. These methods sample independent random variables by computer to estimate distributions, averages, quantiles, roots or optima of functions, etc. They are developed and studied in the abstract framework of probability theory, in which the notion of an infinite sequence of independent random variables uniformly distributed over the interval $(0, 1)$ (i.i.d. $\mathcal{U}(0, 1)$) is well-defined, and the theory is built under the assumption that such random variables can be sampled at will. But the notion of i.i.d. random variables cannot be implemented exactly on current computers. It can be approximated to some extent by physical devices, but these approximations are cumbersome, inconvenient, and not always reliable, so they are rarely used for computational statistics. Random number generators used for Monte Carlo applications are in reality deterministic algorithms whose behavior *imitates* i.i.d. $\mathcal{U}(0, 1)$ random variables. They are pure masquerade. It may be surprising that they work so well, but fortunately they do, or at least some of them do. Here we briefly explain how they are built and tested, and what is the theory behind. Many widely available generators should be avoided and we give examples. We point out reliable ones that can be recommended. More detailed discussions can be found in Knuth (1998); L'Ecuyer (1994, 2006, 2012, 2017); L'Ecuyer et al. (2017, 2021).

Physical devices

Hardware devices such as amplifiers of heat noise in electric resistances, photon counting and photon trajectory detectors, and several others, can produce sequences of random bits, which can in turn be used to construct a sequence of floating-point numbers between 0 and 1 that provides a good approximation of i.i.d. $\mathcal{U}(0, 1)$ random variables. Most of these devices sample a signal at a given frequency and return 1 if the signal is above a given threshold, 0 otherwise. To improve uniformity and reduce the dependence between successive bits of this sequence, the bits can be cleverly combined via simple operations such as exclusive-or and addition modulo 2, to produce a higher-quality sequence, but at a lower frequency Chor and Goldreich (1988). These types of "truly random" sequences are needed for applications such as cryptography and gambling machines, for example, where security and unpredictability are essential. But for Monte

Carlo algorithms and computational statistics in general, adequate statistical behavior can be achieved by much more practical and less cumbersome algorithmic generators, which require no special hardware.

Algorithmic generators

These generators are based on deterministic algorithms that implement transformations which from a given initial state, produce a sequence of output random numbers, usually real numbers in the interval $(0, 1)$ to imitate i.i.d. $\mathcal{U}(0, 1)$ random variables. They are sometimes called *pseudorandom*. For the remainder of this article, a *random number generator* (RNG) means a system with a finite set $\mathcal{S}$ of states, a transition function $f : \mathcal{S} \to \mathcal{S}$ that determines the next state from the current one, and an output function $g : \mathcal{S} \to (0, 1)$ that assigns to each possible state a real number in $(0, 1)$. The system starts from an initial state s_0 (the *seed*) and at each step $i \geq 0$, its output u_i and the next state $s_{i+1} \in \mathcal{S}$ are determined uniquely by the output function and the transition function, respectively. The output values $\{u_i, i \geq 0\}$ are the so-called *random numbers* (an abuse of language) returned by the RNG. In practice, a few truly random bits could be used to select the seed s_0 (although this is rarely done), then everything else is deterministic. In some applications such as for gambling machines in casinos, for example, the state is reseeded frequently with true random bits coming from a physical source, to break the periodicity and determinism. But for Monte Carlo methods, there is no good reason for doing this, so we assume henceforth that no such reseeding is done.

Because the total number of states is finite, the RNG will eventually revisit a state already seen, and from then on the same sequence of states (and output values) will repeat, over and over again. That is, for some $l \geq 0$ and $j > 0$, we have $s_{i+j} = s_i$ and $u_{i+j} = u_i$ for all $i \geq l$. The smallest $j > 0$ that satisfies this condition is the *period length* ρ of the RNG. It can never exceed the total number of states. This means that if the state fits in b bits of memory, the period cannot exceed 2^b. This is not really restrictive, because no current computer can generate more than (say) 2^{128} numbers in a lifetime, and a state of two 64-bit integers would suffice to achieve this.

Compared with generators that exploit physical noise, algorithmic RNGs have the advantage that their sequence can be repeated exactly as many times as we want without storing it. This is convenient for program verification and debugging, and is even more important (crucial, in fact) for key variance reduction methods such as using common random numbers when comparing similar systems, for external control variates, for antithetic variates, for sensitivity analysis (derivative estimation), and for sample-average optimization (Asmussen and Glynn, 2007; L'Ecuyer, 2015, 2023b; Owen, 2023).

Multiple streams and substreams

In modern Monte Carlo software, RNGs are often implemented to offer multiple streams and substreams of random numbers. In object-oriented implementations, a *stream* can be seen as a object that produces a long sequence of i.i.d. $\mathcal{U}(0,1)$ random numbers, and such objects can be created in a practically unlimited number, just like other types of objects. These streams are usually partitioned into substreams and methods (or procedures) are readily available to jump ahead to the next substream, or rewind to the beginning of the current substream, or to the beginning of the whole stream (L'Ecuyer and Côté, 1991; L'Ecuyer et al., 2002; L'Ecuyer and Leydold, 2005; L'Ecuyer, 2015, 2023a). A good implementation must make sure that the streams and substreams are long enough so there is no chance of overlap. A standard way of doing this is to take an RNG with a long-enough period ρ and select two large integers $\rho_2 \ll \rho_1 \ll \rho$. The streams will correspond to blocks of successive values that start ρ_1 steps apart in the RNG sequence, and the substreams will be smaller blocks that start ρ_2 steps apart in the stream. In the `RngStream` implementation of L'Ecuyer et al. (2002), for example, we have $\rho \approx 2^{191}$, $\rho_1 = 2^{127}$, and $\rho_2 = 2^{76}$. In the SSJ library (L'Ecuyer and Buist, 2005; L'Ecuyer, 2023a), a `RandomStream` object represents a stream. Whenever we create a new stream, the software starts from the initial state of the last stream that was created and jumps ahead in the sequence by ρ_1 steps to find the starting point of the new stream. This jumping ahead needs to be done efficiently.

To illustrate why this is useful, suppose we want to simulate two similar systems with well-synchronized common random numbers (Asmussen and Glynn, 2007; Glasserman, 2004; L'Ecuyer, 2023b). Think for instance of a large supply chain model or a queueing network, for which we need to estimate the sensitivity of some performance measure with respect to a small change in the operating policy or in some parameter of the system. We want to simulate the system n times (say), with and without the change, with the same random numbers used for the same purpose (as much as possible) in the two systems. The latter is not always easy to implement, because often, the random numbers are generated in a different order for the two systems, and their required quantity is random and differs across the two systems. For this reason, one would usually create and assign a different random stream for each type of random numbers used in these systems (e.g., each type of arrival, each type of service time, routing decisions at each node, machine breakdowns, etc.) (L'Ecuyer and Buist, 2006; Cezik and L'Ecuyer, 2008; Law, 2014; L'Ecuyer, 2023a). To make sure that the same random numbers from each stream are reused for the two systems for each simulation run, one would simply advance all streams to a new substream at the beginning of a simulation run, simulate the first system, bring these streams back to the beginning of the current substream, simulate the second system, then advance them again to their next substream for the next simulation run. Good simulation and statistical software tools should incorporate

these types of facilities.

Basic requirements

From what we have seen so far, obvious requirements for a good RNG are a very long period, the ability to implement the generator easily in a platform-independent way, the possibility of repeating the same sequence over an over again, the facility of splitting the sequence into several disjoint streams and substreams and jumping across them quickly, and of course good speed for the generator itself. Nowadays, fast generators can produce about one billion $\mathcal{U}(0,1)$ random numbers per second on a laptop computer with a single processor. But these requirements are not sufficient. To see this, consider a RNG that returns $u_i = (i/10^{100}) \mod 1$ at step i. It has all the above properties, but no reasonable statistician would trust it, because of the obvious correlation between the successive outputs. So what else do we need?

Multivariate uniformity

Both uniformity and independence are covered by the following joint statement: For every number of dimensions $s > 0$, the vector of s successive output values $(u_0, \ldots, u_{s-1})$ of the RNG is a random vector with the uniform distribution over the unit hypercube $(0,1)^s$. Of course, this cannot be true for an algorithmic RNG, because there are just a finite number of possibilities for that vector. These possibilities are the vectors in the set $\Psi_s = \{(u_0, \ldots, u_{s-1}) : s_0 \in \mathcal{S}\}$, whose cardinality cannot exceed $|\mathcal{S}|$. For a random initial seed, we basically pick a point at random in Ψ_s as an approximation of picking it at random in $(0,1)^s$. For the approximation to be good, Ψ_s must provide a very even (uniform) coverage of the unit hypercube, for s up to a number as large as possible. Good RNGs are constructed based on a mathematical analysis of this uniformity. A large Ψ_s (i.e., a large $\mathcal{S}$) is needed to provide a good coverage in high dimensions, and this is the main motivation for having a large state space.

There is no universal measure of this uniformity; in practice, the measure is defined differently for different classes of RNGs, depending on their mathematical structure, in a way that it is computable without generating the points explicitly (which would be impossible). This is the main reason why the most popular RNGs are based on linear recurrences: their period length and uniformity can be analyzed much more easily than for nonlinear RNGs. To design an RNG, we first select a construction type that can be implemented in an efficient way, and a size of the state space, then we search for parameters that provide a maximal period given that size and the best possible uniformity of Ψ_s for all s up to a certain preselected threshold. After that, the RNG is implemented and submitted to empirical statistical tests. This type of approach was followed for example by Matsumoto and Nishimura (1998); L'Ecuyer (1999b,a); L'Ecuyer

and Touzin (2000); L'Ecuyer and Granger-Piché (2003); Panneton et al. (2006); Vigna (2017); Blackman and Vigna (2021).

Empirical statistical testing

An unlimited number of statistical tests can be applied to RNGs. These tests take a stream of successive output values and look for evidence against the null hypothesis that they are the realizations of independent $\mathcal{U}(0,1)$ random variables. The hypothesis is rejected when the p-value of the test is extremely close to 0 (which typically indicates strong departure from uniformity) or 1 (which indicates excessive uniformity). As a simple illustration, one might partition the unit hypercube $(0,1)^s$ in k boxes of volume $1/k$, sample n "independent" points at random in $(0,1)^s$ by generating $s\,\mathcal{U}(0,1)$ random variates for each point, and count the number of times C that a point falls in a box already occupied. If both k and n are very large and $\lambda = n^2/(2k)$ is not too large, then C is supposed to behave approximately as a Poisson random variable with mean λ. If c denotes the realization of C, then the p-value can be approximated by the probability that such a Poisson random variable is at least c. If the p-value is much too small (C is much too large), this means that the points tend to fall in the same boxes more often than they should, whereas if it is too large (C is too small), this means that the points fall too rarely in the same boxes. The latter represents a form of excessive uniformity which is also a departure from randomness.

If the outcome is suspicious but unclear (for example, a small p-value but not excessively small), one can reapply the test (independently), perhaps with a larger sample size. Typically, when the suspicious p-value really indicates a problem, increasing the sample size will clarify things rapidly. When problems are detected, it is frequent to find p-values smaller than 10^{-15}, for example. And this happens for many RNGs that have been used in popular software (L'Ecuyer and Simard, 2007).

It is known that constructing an RNG that passes all possible tests is impossible (L'Ecuyer, 2006). The common practice is to forget about the very complicated tests and care only about tests that easy enough to implement. In fact, one could argue that the difference between the good and bad RNGs is that the bad ones fail simple tests whereas the good ones fail only very complicated tests.

Collections of statistical tests for RNGs have been proposed and implemented by Knuth (1998); Marsaglia (1996); L'Ecuyer and Simard (2007, 2013), for example. Statistical tests can never prove that a RNG is defect-free. They can catch some problems and miss others. For this reason, theoretical tests that measure the uniformity by examining the mathematical structure are more important. Empirical tests can improve our confidence in some RNGs and help us discard bad ones, but they should not be taken as the primary selection criterion.

Linear recurrences modulo m

Many good algorithmic RNGs found in simulation software have a transition function defined by a linear recurrence of the type

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

for some positive integers k and m, and coefficients $a_1, \ldots, a_k$ in $\{0, 1, \ldots, m-1\}$, with $a_k \neq 0$. This recurrence can also be written in matrix form as $\boldsymbol{x}_i = \boldsymbol{A}\boldsymbol{x}_{i-1} \mod m$ where $\boldsymbol{x}_i = (x_{i-k+1}, \ldots, x_i)^\intercal$ for a matrix $\boldsymbol{A}$ with ones just above the diagonal and the a_j's in the last row. One can obtain the period length $m^k - 1$ by taking m as a prime number and choosing the coefficients a_j appropriately (L'Ecuyer, 1990; Niederreiter, 1992; Knuth, 1998). The output can be defined as $u_i = x_i/m$, or $u_i = (x_i + 1)/(m + 1)$, or $u_i = (x_i + 1/2)/m$, for example. This type of RNG is known as a *multiple recursive generator* (MRG). For $k = 1$, we obtain the classical (but obsolete) *linear congruential generator* (LCG). It is easy to advance the state of the MRG by an arbitrary number of steps in a single large jump: $\boldsymbol{x}_{i+\nu} = (\boldsymbol{A}^\nu \mod m)\boldsymbol{x}_i \mod m$, after $\boldsymbol{A}^\nu \mod m$ has been precomputed (L'Ecuyer, 1990).

The uniformity of Ψ_s for the MRG can be measured by exploiting its lattice structure. Figures of merit (related to the so-called spectral test) to measure the quality of this lattice are proposed in (Knuth, 1998; L'Ecuyer, 1997, 2012).

Typically, m is a prime number that fits the 32-bit or 64-bit word of the computer and the multipliers a_j are chosen so that the recurrence can be computed very quickly. But the quest for speed often goes too far. For example, popular types of generators known as lagged-Fibonacci, add-with-carry, and subtract-with-borrow (which are slight modifications of the MRG) employ only two nonzero coefficients, say a_r and a_k, both equal to ± 1. It turns out that all triples of the form (u_i, u_{i-r}, u_{i-k}) produced by these generators lie in only two parallel planes in the three-dimensional unit cube (L'Ecuyer, 1997). These generators have already given totally wrong results in real-life Monte Carlo applications and should not be used. LCGs with modulus $m \leq 2^{64}$ should also be discarded, because their state space is too small.

An effective construction technique for good MRGs is to combine (say) two or three of them, for example by adding their outputs modulo 1. The idea is to select the components so that a fast implementation is available, while the combined MRG has a long period and its point set Ψ_s has good uniformity. Good parameters can be found by extensive computer searches. Specific constructions of this type were made by L'Ecuyer (1999a); L'Ecuyer and Touzin (2000), for example. The `RngStream` system of L'Ecuyer et al. (2002) is based on the `MRG32k3a` generator, which is of this type.

Linear recurrences modulo 2

Given that computers work in binary arithmetic, it is no surprise that many of the fastest good RNGs are based on linear recurrences modulo 2. That is, recurrence (1) with $m = 2$. This can be framed in matrix notation (L'Ecuyer, 2006; L'Ecuyer and Panneton, 2009; L'Ecuyer, 2023b) as:

$$\begin{aligned} \boldsymbol{x}_i &= \boldsymbol{A}\boldsymbol{x}_{i-1} \bmod 2, \\ \boldsymbol{y}_i &= \boldsymbol{B}\boldsymbol{x}_i \bmod 2, \\ u_i &= \sum_{\ell=1}^{w} y_{i,\ell-1} 2^{-\ell} \end{aligned}$$

where $\boldsymbol{x}_i = (x_{i,0}, \ldots, x_{i,k-1})^{\mathsf{t}}$ is the k-bit *state vector* at step i, $\boldsymbol{y}_i = (y_{i,0}, \ldots, y_{i,w-1})^{\mathsf{t}}$ is a w-bit *output vector*, k and w are positive integers, $\boldsymbol{A}$ is a $k \times k$ binary matrix, $\boldsymbol{B}$ is a $w \times k$ binary matrix, and $u_i \in [0, 1)$ is the *output* at step i. In practice, the output can be modified slightly to make sure that the generator never returns exactly 0.

Many popular types of generators belong to this class, including the Tausworthe or linear feedback shift register (LFSR) generator, polynomial LCG, generalized feedback shift register (GFSR), twisted GFSR, Mersenne twister, WELL, xorshift, xoshiro, xoroshiro, linear cellular automaton, and combinations of these (Matsumoto and Nishimura, 1998; L'Ecuyer, 1999b, 2006; L'Ecuyer and Panneton, 2005, 2009; Panneton and L'Ecuyer, 2005; Panneton et al., 2006; Blackman and Vigna, 2021). The largest possible period is $2^k - 1$, reached when the characteristic polynomial of $\boldsymbol{A}$ is a primitive polynomial modulo 2. The matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ are always selected to allow a fast implementation by using just a few simple binary operations such as or, exclusive-or, shift, and rotation, on blocks of bits, while still providing good uniformity for the point set Ψ_s. This uniformity is assessed by measures of equidistribution of the points in the diadic rectangular boxes obtained by partitioning $(0, 1)$ for each axis j into intervals of lengths 2^{-q_j} for some integers $q_j \geq 0$ (L'Ecuyer and Panneton, 2009). Combined generators of this type, obtained by a bitwise exclusive-or of the output vectors $\boldsymbol{y}_i$ of two or more generators from that class, are equivalent to yet another generator from the same class (L'Ecuyer, 1996, 1999b; L'Ecuyer and Panneton, 2009). Their motivation is the same as for combined MRGs. For the xoshiro and xoroshiro of Blackman and Vigna (2021), a simple nonlinear scramble is applied to the output to break the linearity modulo 2.

Nonlinear generators

Linear RNGs have a regular structure that can eventually be detected by statistical tests specifically designed to detect this structure (L'Ecuyer and Simard, 2007). Cryptologists know well about that and use (slower) nonlinear RNGs for that reason. For Monte Carlo, the linearity itself is practically never a problem, because the random numbers

are almost always transformed in a nonlinear way by the simulation algorithm. But there are situations where linearity matters. For example, if we generate large random binary matrices and the rank of the matrix must have the right distribution, then we should not use a linear generator modulo 2, because there are too many linear dependencies between the bits (L'Ecuyer and Simard, 2007).

A nonlinear RNG is easily obtained by adding a nonlinear output transformation to a linear RNG, or by shuffling its output values using another generator, or by using a nonlinear recurrence in the transition function, or by combining two generators of different types, such as an MRG and a generator based on a linear recurrence modulo 2. For nonlinear RNGs, the uniformity of Ψ_s is generally too difficult to analyze. But for the last type of combination just mentioned, useful bounds can be obtained on uniformity measures (L'Ecuyer and Granger-Piché, 2003). It is also important to understand that combining generators does not necessarily leads to an improvement. Nonlinear RNGs are also slower in general than their linear cousins. On the other hand, they tend to perform better in empirical statistical tests (L'Ecuyer and Simard, 2007).

Counter-based generators

With recurrence-based RNGs as we have seen so far, if we want to create several thousands streams all at one time, for example to make computations on a GPU, we need to jump ahead by ρ_1 steps in the sequence for each new stream, and this is an inherently sequential process. Creating the streams in parallel is generally difficult because it means jumping ahead by different multiples of ρ_1. *Counter-based generators* avoid this problem by making the transition function f extremely simple: it just increases a counter by 1. To jump ahead by ν steps, it suffices to increase the counter by ν. All the complicated work is left to the output function g, which is usually taken as a simplified cryptographic hashing function. The state is actually a pair (k, n) where k is an m-bit integer called the *key* and n is a c-bit *counter* which starts at 0 and increases by 1 modulo 2^c at each step. One can have $m = c = 128$, for example. The function g should be selected so that for any given k, the set $\{g(k, n) : 0 \leq n < 2^c\}$ covers the interval $(0, 1)$ very evenly. Multiple streams are usually defined by assigning one stream to each value of the key. Creating in parallel thousands or even millions of streams is then trivial.

This type of generator corresponds to the CTR (counter) mode of operation for block ciphers in cryptography (Dworkin, 2001; Daemen and Rijmen, 2002). Hellekalek and Wegenkittl (2003) studied a version based on AES, which performs very well in statistical tests, but is rather slow. Salmon et al. (2011) proposed faster ones named ARS, Threefry, and most notably Philox, which is well-adapted to GPUs. See L'Ecuyer et al. (2021) for more discussion.

Recommendations

When asked for recommendations on uniform RNGs, my natural response is to tell which ones I use for my own experiments. The ones I use most of the time are `MRG32k3a` from L'Ecuyer (1999a), `MRG31k3p` from L'Ecuyer and Touzin (2000), and `LFSR113` and `LFSR258` from L'Ecuyer (1999b). These RNGs are robust and reliable, based on a solid theoretical analysis, and also provide multiple streams and substreams (L'Ecuyer et al., 2002; L'Ecuyer, 2023a). The first three were designed for 32-bit computers and the last one is in 64 bits. The LFSRs are faster than the MRGs, but they have a linear structure modulo 2, as mentioned earlier. The xoshiro and xoroshiro of Blackman and Vigna (2021) are also fast and reliable and do not have this linear structure, thanks to their nonlinear output transformation. The Mersenne twister `MT19937` of Matsumoto and Nishimura (1998) and the WELL generators of Panneton et al. (2006) are also fast and reliable, but they have a huge state, so using them for multiple streams is inefficient. Philox from Salmon et al. (2011) can be a good choice for GPUs.

A list of generators that should be discarded would be much longer; see for example L'Ecuyer (1997); L'Ecuyer and Simard (2007, 2014), and L'Ecuyer et al. (2020). Do not trust blindly the software vendors. Check the default RNG of your favorite software and be ready to replace it if needed. This last recommendation was made over and over again in the past 50 years, and it remains as relevant today as 50 years ago.

Acknowledgment

This work has been supported by the Natural Sciences and Engineering Research Council of Canada Discovery Grant RGPIN-2018-05795 and a Canada Research Chair to the author.

About the author

Pierre L'Ecuyer is a Professor in Computer Science and Operations Research at the University of Montreal. He published 296 scientific articles, book chapters, and books in various areas related to stochastic modeling and simulation. He received prestigious prizes such as the INFORMS Simulation Society Professional Lifetime Achievement Award in 2020, Computer Science Canada Lifetime Achievement Award in 2019, ACM SIGSIM Distinguished Contributions Award in 2016, Canadian Operational Research Society Award of Merit in 2014, INFORMS Simulation Society Distinguished Service Award in 2011, and the INFORMS Simulation Society Outstanding Research Publication Award three times, in 1999, 2009, and 2018. He has been Editor-in-Chief of the *ACM Transactions on Modeling and Computer Simulation*, editor for six other journals, and a referee for 171 different scientific journals.

References

Asmussen, S. and Glynn, P. W. (2007). *Stochastic Simulation*. Springer-Verlag, New York.

Blackman, D. and Vigna, S. (2021). Scrambled linear pseudorandom number generators. *ACM Transactions on mathematical Software*, 47(4):Article 36.

Ceŗik, M. T. and L'Ecuyer, P. (2008). Staffing multiskill call centers via linear programming and simulation. *Management Science*, 54(2):310–323.

Chor, B. and Goldreich, O. (1988). Unbiased bits from sources of weak randomness and probabilistic communication complexity. *SIAM Journal on Computation*, 17(2):230–261.

Daemen, J. and Rijmen, V. (2002). *The Design of Rijndael*. Springer Verlag, New York, NY.

Dworkin, M. (2001). Recommendation for block cipher modes of operation: Methods and techniques. NIST-SP-800-38a, U.S. DoC/National Institute of Standards and Technology.

Glasserman, P. (2004). *Monte Carlo Methods in Financial Engineering*. Springer-Verlag, New York.

Hellekalek, P. and Wegenkittl, S. (2003). Empirical evidence concerning AES. *ACM Transactions on Modeling and Computer Simulation*, 13(4):322–333.

Knuth, D. E. (1998). *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, third edition.

Law, A. M. (2014). *Simulation Modeling and Analysis*. McGraw-Hill, New York, fifth edition.

L'Ecuyer, P. (1990). Random numbers for simulation. *Communications of the ACM*, 33(10):85–97.

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

L'Ecuyer, P. (1996). Maximally equidistributed combined Tausworthe generators. *Mathematics of Computation*, 65(213):203–213.

L'Ecuyer, P. (1997). Bad lattice structures for vectors of non-successive values produced by some linear recurrences. *INFORMS Journal on Computing*, 9(1):57–60.

L'Ecuyer, P. (1999a). Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164.

L'Ecuyer, P. (1999b). Tables of maximally equidistributed combined LFSR generators. *Mathematics of Computation*, 68(225):261–269.

L'Ecuyer, P. (2006). Uniform random number generation. In Henderson, S. G. and Nelson, B. L., editors, *Simulation*, Handbooks in Operations Research and Management Science, pages 55–81. Elsevier, Amsterdam, The Netherlands. Chapter 3.

L'Ecuyer, P. (2012). Random number generation. In Gentle, J. E., Haerdle, W., and Mori, Y., editors, *Handbook of Computational Statistics*, pages 35–71. Springer-Verlag, Berlin, second edition.

L'Ecuyer, P. (2015). Random number generation with multiple streams for sequential and

parallel computers. In *Proceedings of the 2015 Winter Simulation Conference*, pages 31–44. IEEE Press.

L'Ecuyer, P. (2017). History of uniform random number generation. In *Proceedings of the 2017 Winter Simulation Conference*, pages 202–230. IEEE Press.

L'Ecuyer, P. (2023a). SSJ: Stochastic simulation in Java. `https://github.com/umontreal-simul/ssj`.

L'Ecuyer, P. (2023b). Stochastic simulation and Monte Carlo methods. Draft Textbook, `https://www-labs.iro.umontreal.ca/~lecuyer/ift6561/book.pdf`.

L'Ecuyer, P. and Buist, E. (2005). Simulation in Java with SSJ. In *Proceedings of the 2005 Winter Simulation Conference*, pages 611–620. IEEE Press.

L'Ecuyer, P. and Buist, E. (2006). Variance reduction in the simulation of call centers. In *Proceedings of the 2006 Winter Simulation Conference*, pages 604–613. IEEE Press.

L'Ecuyer, P. and Côté, S. (1991). Implementing a random number package with splitting facilities. *ACM Transactions on Mathematical Software*, 17(1):98–111.

L'Ecuyer, P. and Granger-Piché, J. (2003). Combined generators with components from different families. *Mathematics and Computers in Simulation*, 62:395–404.

L'Ecuyer, P. and Leydold, J. (2005). `rstream`: Streams of random numbers for stochastic simulation. *The R Newsletter*, 5(2):16–19.

L'Ecuyer, P., Munger, D., Oreshkin, B., and Simard, R. (2017). Random numbers for parallel computers: Requirements and methods, with emphasis on GPUs. *Mathematics and Computers in Simulation*, 135:3–17.

L'Ecuyer, P., Nadeau-Chamard, O., Chen, Y.-F., and Lebar, J. (2021). Multiple streams with recurrence-based, counter-based, and splittable random number generators. In *Proceedings of the 2021 Winter Simulation Conference*, pages 1–16. IEEE Press.

L'Ecuyer, P. and Panneton, F. (2005). Fast random number generators based on linear recurrences modulo 2: Overview and comparison. In *Proceedings of the 2005 Winter Simulation Conference*, pages 110–119. IEEE Press.

L'Ecuyer, P. and Panneton, F. (2009). $\mathbf{F}_2$-linear random number generators. In Alexopoulos, C., Goldsman, D., and Wilson, J. R., editors, *Advancing the Frontiers of Simulation: A Festschrift in Honor of George Samuel Fishman*, pages 169–193. Springer-Verlag, New York.

L'Ecuyer, P. and Simard, R. (2007). TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4):Article 22.

L'Ecuyer, P. and Simard, R. (2013). *TestU01: A Software Library in ANSI C for Empirical Testing of Random Number Generators; User's Guide*. Software user's guide, version of May 16, 2013, Available at `http://simul.iro.umontreal.ca/testu01`.

L'Ecuyer, P. and Simard, R. (2014). On the lattice structure of a special class of multiple recursive random number generators. *INFORMS Journal on Computing*, 26(2):449–460.

L'Ecuyer, P., Simard, R., Chen, E. J., and Kelton, W. D. (2002). An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50(6):1073–1075.

L'Ecuyer, P. and Touzin, R. (2000). Fast combined multiple recursive generators with multipliers of the form $a = \pm 2^q \pm 2^r$. In *Proceedings of the 2000 Winter Simulation Conference*, pages 683–689. IEEE Press.

L'Ecuyer, P., Wambergue, P., and Bourceret, E. (2020). Spectral analysis of the MIXMAX random number generators. *INFORMS Journal on Computing*, 32(1):135–144.

Marsaglia, G. (1996). DIEHARD: a battery of tests of randomness. See `http://www.stat.fsu.edu/pub/diehard`.

Matsumoto, M. and Nishimura, T. (1998). Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30.

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

Owen, A. B. (2023). *Monte Carlo theory, methods and examples.* `https://artowen.su.domains/mc/`.

Panneton, F. and L'Ecuyer, P. (2005). On the xorshift random number generators. *ACM Transactions on Modeling and Computer Simulation*, 15(4):346–361.

Panneton, F., L'Ecuyer, P., and Matsumoto, M. (2006). Improved long-period generators based on linear recurrences modulo 2. *ACM Transactions on Mathematical Software*, 32(1):1–16.

Robert, C. P. and Casella, G. (2004). *Monte Carlo Statistical Methods*. Springer-Verlag, New York, NY, second edition.

Salmon, J. K., Moraes, M. A., Dror, R. O., and Shaw, D. E. (2011). Parallel random numbers: as easy as 1, 2, 3. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 16:1–16:12, New York. Association for Computing Machinery.

Vigna, S. (2017). Further scramblings of Marsaglia's xorshift generators. *Journal of Computational and Applied Mathematics*, 315:175–181.