Proceedings of the 2015 Winter Simulation Conference
L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, eds.

RANDOM NUMBER GENERATION WITH MULTIPLE STREAMS FOR SEQUENTIAL AND PARALLEL COMPUTING

Pierre L'Ecuyer

DIRO, Pavillon Aisenstadt, Université de Montreal
C.P. 6128, Succ. Centre-Ville
Montréal (Québec), H3C 3J7, CANADA
and Inria Rennes Bretagne–Atlantique, FRANCE

ABSTRACT

We provide a review of the state of the art on the design and implementation of random number generators (RNGs) for simulation, on both sequential and parallel computing environments. We focus on the need for multiple independent streams and substreams of random numbers, explain how they can be constructed and managed, review software libraries that offer them, and illustrate their usefulness via examples. We also review the basic quality criteria for good random number generators and their theoretical and empirical testing.

1 INTRODUCTION

RNGs used for simulation are deterministic algorithms that just *imitate*, to a certain extent, the realizations of independent random variables having the uniform distribution over the interval $(0,1)$ (i.i.d. $\mathscr{U}(0,1)$ for short), or over a certain range of integer values (L'Ecuyer 1994, Knuth 1998, L'Ecuyer 2012). These "random numbers" serve as the basic ingredients to simulate randomness. They are often transformed to simulate random variables from non-uniform distributions, e.g., by applying the inverse of the cumulative distribution function (cdf), and to simulate random vectors, stochastic processes, etc. (Devroye 1986, Hörmann, Leydold, and Derflinger 2004, Asmussen and Glynn 2007, Law 2014).

These deterministic algorithms are much more convenient for simulation than genuinely random numbers produced by physical devices such as thermal noise diodes, photon trajectory detectors, etc., because they make it easy to reproduce exactly the same sequence as many times as we want. This reproducibility is essential in many simulation applications, such as for program verification and debugging, and for comparing similar systems with common random numbers (CRNs) to reduce the variance of the difference in performance (L'Ecuyer 2007, L'Ecuyer 2008, Passerat-Palmbach, Mazel, and Hill 2012, Law 2014, L'Ecuyer, Munger, and Kemerchou 2015b). We want a given simulation program to produce exactly the same results when we run it again on the same computer or a different computer, including parallel computers with different architectures.

Multiple independent streams of random numbers that can evolve independently of each other are essential for parallel simulation (P. L'Ecuyer and D. Munger and B. Oreshkin and R. Simard 2015). They are also useful for simulation on a single processor, in particular to synchronize the random numbers when using CRNs to compare systems with slightly different configurations or different decision making policies (L'Ecuyer et al. 2002, L'Ecuyer 2007, Law 2014), to make sure that exactly the same random numbers are used for exactly the same purpose in all configurations or for all policies. Typically, one wishes to perform say n independent replications of each configuration, with CRNs across configurations. It is then convenient to have streams divided into segments of equal length called substreams, as in L'Ecuyer and Côté (1991) and L'Ecuyer et al. (2002). One can use a separate stream for each source of random numbers needed in the system, and one different substream for each replication.

The easiest (and usual) way to provide such streams and ensure reproducibility is via a central monitor that creates and manages the streams. Reliable software to do that has been available for several years; for example RngStreams (L'Ecuyer et al. 2002) and SSJ (L'Ecuyer 2008). This software was designed mainly for simulation over a sequential computer with a single processor, but it can also be used directly in a parallel processing environment, provided that only the (single) central monitor is allowed to create new streams. Karl et al. (2014) show how to use RngStreams without change in both OpenMP and MPI, which are the most popular parallel programming standards for shared memory and message-passing environments, respectively.

Graphical processing units (GPUs) are special computing devices originally designed for image-rendering on computer screens. In recent years, we saw an increasing interest for their usage to accelerate computations at low-cost for other applications, including scientific computing and simulation. GPUs execute groups of (typically 32 or 64) parallel *threads* (or *work items*) in a *single instruction multiple data* (SIMD) mode, which means that all the threads in each group must execute exactly the same instructions in parallel at each time step, on an array of data (Khronos Group 2010, NVIDIA 2015, L'Ecuyer, Munger, and Kemerchou 2015a). The reason for this restriction is to reduce the work (and time) required for task scheduling. Moreover, each thread has fast access to only a small amount of private memory. These restrictions must be taken into account in the design of algorithms that run on such devices. In particular, the most popular RNGs currently available have been designed for a single processor with a large memory, and many of them (such as the Mersenne twister and WELL RNGs, for example), have a large state that requires significant memory space to store, which makes them inappropriate for GPU computations in which each stream runs on a single processing element (in a single thread) at a time. RNGs with a small state and that are appropriate for GPUs have been proposed recently.

The rest of this paper is organized as follows. In Section 2, we recall the definition of algorithmic RNG, the requirements and selection criteria for good RNGs, and their theoretical and empirical testing. In Section 3 we review the most popular types of RNG constructions used for simulation. In Section 4, we explain how to make multiple streams and manage them, using a single monitor. In Section 5, we give an example to illustrate the usefulness of multiple streams and substreams, both on a single processor and on multiple processors.

Most of the material presented here is drawn from various earlier articles and reports from the author, including L'Ecuyer (1990), L'Ecuyer (1994), L'Ecuyer (2012), L'Ecuyer, Munger, and Kemerchou (2015b), and P. L'Ecuyer and D. Munger and B. Oreshkin and R. Simard (2015).

2 ALGORITHMIC RNGS AND SELECTION CRITERIA

L'Ecuyer (1994) defines an algorithmic RNG as a deterministic automata with a finite set of *states* $\mathcal{S}$ (the state space), a transition function $f : \mathcal{S} \to \mathcal{S}$, an output function $g : \mathcal{S} \to (0,1)$, and an initial state (or *seed*) s_0. The seed s_0 can be chosen randomly or not. The state evolves according to $s_i = f(s_{i-1})$. The output returned at step i is $u_i = g(s_i) \in (0,1)$, for $i \geq 0$. The sequence of states s_i is periodic and the period length ρ, defined as the smallest $j > 0$ such that $s_{l+j} = s_l$ for some $l \geq 0$, cannot exceed $|\mathcal{S}|$, the (finite) cardinality of the state space. Well-designed RNGs usually have $l = 0$ and ρ close to $2^k \geq |\mathcal{S}|$, where k is the number of bits used to represent the state.

Quality criteria for RNGs include high running speed, long period (e.g., 2^{200} or more), exact repeatability on various computing platforms, ease of implementation in a platform-independent way, and the availability of an efficient method to split the sequence into long disjoint streams and to jump quickly between them. An even more important requirement is for successive output values to behave as independent $\mathcal{U}(0,1)$ random variables. This can be approximated as follows: If we select the seed s_0 randomly and uniformly over $\mathcal{S}$, the s-dimensional vector $\mathbf{u}_{0,s} = (u_0, \ldots, u_{s-1})$ should have (approximately) the uniform distribution over $(0,1)^s$, for any $s > 0$. But in fact, this vector has exactly the uniform distribution over the finite set

$$\Psi_s = \{(u_0, \ldots, u_{s-1}) = (g(s_0), \ldots, g(s_{s-1})) : s_0 \in \mathcal{S}\}.$$

From this perspective, we want Ψ_s to cover the unit hypercube $(0,1)^s$ as evenly as possible, in some sense. Good RNGs are usually constructed based on a mathematical analysis of this uniformity, which is measured by a figure of merit that can be computed efficiently even when $\mathscr{S}$ is huge. These measures depend on the structure of the RNG; they include the spectral test and measures of equidistribution (Knuth 1998, L'Ecuyer and Panneton 2009, L'Ecuyer 2012). A larger Ψ_s (larger $\mathscr{S}$) can potentially cover $(0,1)^s$ more evenly, but a large $\mathscr{S}$ and large period are not sufficient for high quality. A large $\mathscr{S}$ (and large state) also has drawbacks: it requires more memory to store and manipulate the state, and more overhead to compute the starting points of multiple streams and to copy their states. On GPU devices, this can be unacceptable.

After an appropriate mathematical analysis of the period and multivariate structure, a proposed RNG should be submitted to empirical statistical tests of uniformity and independence (Knuth 1998, L'Ecuyer and Simard 2007). As an example of a statistical test, one can partition the unit hypercube $(0,1)^s$ into $k = 2^{ds}$ subcubes of equal size, for some integers s and d, then generate n points (vectors) $\mathbf{u} = (u_0, \ldots, u_{s-1})$, check in what subcube each vector falls, and count the number of times C a point falls in a subcube in which another point already fell previously. When n is large and $n^2/(2k)$ is not too large, under the null hypothesis $\mathscr{H}_0$ that the RNG produces independent $\mathscr{U}(0,1)$ random variables (so the points are truly uniform and independent), this random variable C has approximately a Poisson distribution with mean $\lambda = n^2/(2k)$ (L'Ecuyer, Simard, and Wegenkittl 2002). To test if an observed number c of collisions "agrees" with this theoretical distribution, one can compute the right and left p-values, defined as $p^+ = \mathbb{P}[X \geq c]$ and $p^- = \mathbb{P}[X \leq c]$ where X is a Poisson random variable with mean λ. If one of those is very small, for example less than 10^{-10} or less than 10^{-15} as in L'Ecuyer and Simard (2007), we have found evidence against $\mathscr{H}_0$. A very small p^+ indicates a lack of uniformity (too many collision), whereas a very small p^- indicates excessive uniformity, which means a lack of independence (two few collisions). In either case, the RNG fails the test. This test can actually be repeated r times independently, i.e., on disjoint segments of the RNG cycle, and the empirical distribution of the r realizations of C can be compared with the Poisson(λ) distribution via some goodness-of-fit test, which also produces a p-value. This is a *two-level test*.

This *collision test* is only an example; there is in fact an unlimited number of ways of defining a statistical test for RNGs. Each test computes a p-value, and the RNG can be declared to fail the test when this p-value is much too close to 0 or 1. When the distribution of the test statistic under $\mathscr{H}_0$ is not continuous, we can distinguish the positive and negative p-values, as we did above. When the p-value is small but not extremely small, e.g., around 10^{-3} to 10^{-8}, we can just repeat the test with a different and larger segment of the cycle, until the result is clear.

The TestU01 software (L'Ecuyer and Simard 2007) contains the most extensive collection of statistical tests currently available for RNGs. Aside from the long list of individual tests that can be parameterized at will, it offers predefined batteries of tests called SmallCrush, Crush, BigCrush, and Rabbit, which are very popular. RNGs that pass the three Crush batteries are called *Crush-resistant* (Salmon et al. 2011). These Crush batteries are for sequences of real numbers in $(0,1)$ that are supposed to imitate i.i.d. $\mathscr{U}(0,1)$ random variables, whereas Rabbit tests binary sequences which are assumed to be independent random bits.

Of course, one can only apply a finite and small number of tests in practice, and this can never prove that a RNG is flawless. We think that theoretical tests that measure the uniformity by studying the mathematical structure are more important than empirical tests, when they can be applied.

3 POPULAR TYPES OF RNGS FOR SIMULATION

3.1 Linear RNGs

The most common RNGs used for simulation have a state that evolves according to a linear recurrence modulo some integer $m \geq 2$ (L'Ecuyer 1990, Niederreiter 1992, L'Ecuyer 1994, Knuth 1998, L'Ecuyer 2012): The state at step i can be represented as a k-dimensional vector $\mathbf{x}_i = (x_{i,0}, \ldots, x_{i,k-1})^\mathsf{T}$ with components

in $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$ and the transition function f is defined by

$$\mathbf{x}_i \;=\; \mathbf{A}\mathbf{x}_{i-1} \bmod m, \tag{1}$$

where $\mathbf{A}$ is a $k \times k$ matrix with elements in $\mathbb{Z}_m$. The largest period that can be obtained by this construction is $m^k - 1$: all m^k possible vectors $\mathbf{x} \in \mathbb{Z}_m^k$ except the zero vector $\mathbf{x} = \mathbf{0}$ are visited exactly once over a cycle. This occurs if and only if m is a prime number, so $\mathbb{Z}_m$ can be identified with the finite field $\mathbb{F}_m$ with m elements, and the characteristic polynomial of $\mathbf{A}$ is a primitive polynomial over $\mathbb{F}_m$. The most popular linear constructions satisfy these conditions. The matrix $\mathbf{A}$ is also selected in a way that a very fast implementation of (1) is available (i.e., with very few elementary operations on a computer). For this reason, $\mathbf{A}$ is usually a sparse matrix. Typical values of m are $m = 2$ (so we work in binary arithmetic, which can be very fast), and prime numbers slightly smaller than 2^{31} (for 32-bit integer arithmetic) or than 2^{63} (for 64-bit arithmetic). Below, we will discuss those two situations separately.

Given the linear recurrence (1), if k is not too large, one can quickly jump ahead by ν steps, from $\mathbf{x}_i$ to $\mathbf{x}_{i+\nu}$, for an arbitrarily large positive integer ν, via the simple matrix-vector multiplication

$$\mathbf{x}_{i+\nu} = (\mathbf{A}^\nu \bmod m)\mathbf{x}_i \bmod m, \tag{2}$$

where $\mathbf{A}^\nu \bmod m$ can be precomputed in advance (L'Ecuyer 1990, L'Ecuyer 2012, L'Ecuyer et al. 2002). This is very handy to compute the initial states of successive "disjoint" streams of random numbers that start ν steps apart in the original sequence, for some $\nu \geq 2^{100}$, for example.

One must also define the output function g, which in the linear case takes the following form:

$$\mathbf{y}_i \;=\; \mathbf{B}\mathbf{x}_i \bmod m, \tag{3}$$

$$u_i \;=\; \sum_{\ell=1}^{w} y_{i,\ell-1} m^{-\ell} \tag{4}$$

in which $\mathbf{y}_i = (y_{i,0}, \ldots, y_{i,w-1})^\mathsf{T} \in \mathbb{Z}_m^w$ for some integer $w \geq 1$, $\mathbf{B}$ is a $w \times k$ matrix over $\mathbb{Z}_m$, $\ell \geq 1$ is an integer, and $u_i \in [0, 1)$ is the *output* at step i. In practice, the output function is modified slightly to make sure it never returns 0, e.g., by adding by adding $(2m)^{-w}$. Typical values of ℓ are $\ell = 31$ or 32 or 64 when $m = 2$ and $\ell = 1$ when m is large.

3.2 Linear Recurrences Modulo 2

Most of the fastest RNGs for simulation are linear modulo 2 (i.e., over $\mathbb{F}_2$). This includes the linear feedback shift register (LFSR) generator, generalized feedback shift register (GFSR), twisted GFSR, Mersenne twister (MT), WELL generators, xorshift generators, shift registers in lookup tables (LUT-SR), and combinations of these; see L'Ecuyer (1996b), Matsumoto and Nishimura (1998), L'Ecuyer (1999b), Panneton et al. (2006), L'Ecuyer and Panneton (2009), L'Ecuyer (2012), Thomas and Luk (2013). The maximal period is $2^k - 1$. The matrices $\mathbf{A}$ and $\mathbf{B}$ usually represent simple operations on blocks of bits such as AND, OR, exclusive-OR, shift, rotation, etc., that are fast to execute. They are also selected so that the sets Ψ_s have good uniformity over some range of values of s. The uniformity is evaluated by verifying the equidistribution of the points over a collection of dyadic rectangular boxes (L'Ecuyer 1996b, L'Ecuyer 1999b, L'Ecuyer and Panneton 2009). This is achieved efficiently (without generating the points) by exploiting the linear structure.

$\mathbb{F}_2$-linear RNGs have the property (among others) that for any given bit of the output, say for bit ℓ, the sequence $\{y_{i,l}, i \geq 0\}$ follows a linear recurrence or order k with the same characteristic polynomial as $\mathbf{A}$. This linear behavior in the output is easily detected by a statistical test that measures the linear complexity of this binary sequence, by computing the shortest order of a recurrence that this sequence obeys. Since the Crush and Rabbit batteries in TestU01 contain such tests, none of the $\mathbb{F}_2$-linear RNGs is Crush-resistant, because they all fail this linear complexity test. They also fail tests on the rank of "random" binary matrices constructed with blocks of successive bits from the RNGs (also included in BigCrush), due to the linear

relationships between those bits. Nevertheless, these RNGs are still appropriate and safe when the linear relationship between the bits has no impact (e.g., when the bits are transformed nonlinearly to produce the output), which is the case for most applications

Notable instances of large $\mathbb{F}_2$-linear RNGs that can be recommended include the *Mersenne twister* (MT) `MT19937` (Matsumoto and Nishimura 1998) and the WELL generators (Panneton, L'Ecuyer, and Matsumoto 2006). They both have implementations with a primitive polynomial of order $k = 19937$. The WELL also has implementations for other values of k, smaller and larger, up to $k = 44497$. The WELL RNGs have been designed as an improvement over the MT. They change more bits of the state at each iteration and their state moves more rapidly (in some sense) in the state space. See Panneton, L'Ecuyer, and Matsumoto (2006) for further explanations. These RNGs use very large values of k, so they have an extremely large period, but also a large state (k bits), much too large than necessary. This makes them inconvenient for GPUs, in particular, because the state occupies way too much memory. Such a very large k also makes the jumping-ahead via (2) very slow (Haramoto et al. 2008).

If we combine two or more $\mathbb{F}_2$-linear RNGs via a bitwise xor of their output vectors $\mathbf{y}_i$, the resulting output sequence is equivalent to that of another $\mathbb{F}_2$-linear RNG whose characteristic polynomial is the product of those of the components. Its period can equal the product of the periods of the components, if the latter have no common factor. The measures of equidistribution can be computed in a similar way as in the single-component case. This provides an effective way to construct RNGs with much larger periods and better behaved point sets Ψ_s than their components (L'Ecuyer and Panneton 2009), and whose recurrence can be implemented by implementing the recurrences of the (smaller) components separately. This can be efficient because the components can have much smaller values of k. Specific instances that can be recommended are LFRS113 and LFSR258, two combined LFSR generators proposed by L'Ecuyer (1999b), with period lengths near 2^{113} and 2^{258}, respectively. Efficient implementations with multiple streams and substreams are easily done; are available for LFSR113 in L'Ecuyer (2008) and L'Ecuyer, Munger, and Kemerchou (2015b).

3.3 Linear Recurrences Modulo $m > 2$

When m is large, we often use a recurrence of the form

$$x_i \;=\; (a_1 x_{i-1} + \cdots + a_k x_{i-k}) \bmod m, \tag{5}$$

for some coefficients $a_1, \ldots, a_k$ in $\mathbb{Z}_m$, with $a_k \neq 0$. This can be written as (1) if we take $\mathbf{x}_i = (x_{i-k+1}, \ldots, x_i)^\mathsf{T}$ and

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ a_k & a_{k-1} & \cdots & a_1 \end{pmatrix}.$$

Taking $w = 1$ gives the output $u_i = x_i/m$. This is known as a *multiple recursive generator* (MRG). For $k = 1$, it gives the now obsolete *linear congruential generator* (LCG), whose period much too small for current computers and applications. All popular LCGs from the past fail the BigCrush battery. Taking k too large, on the other hand, makes the state too large and reduces efficiency (to store and copy the state, to jump ahead, etc.). In practice, one would take for example $u_i = (x_i + 1)/(m+1)$ or $u_i = (x_i + 1/2)/m$, to avoid returning 0. This has no impact on the period and affects only marginally the structure of the point sets Ψ_s.

The uniformity of Ψ_s is typically measured via the *spectral test*, which exploits the fact that Ψ_s has a lattice structure and measures the distance between successive equidistant parallel hyperplanes that contains all the points of Ψ_s (L'Ecuyer and Couture 1997, Knuth 1998, L'Ecuyer 1999a). We want this distance to be small and we usually transform it to a normalized figure of merit by dividing the best possible value (or a lower bound) by the actual distance.

The multipliers a_j are selected so that (5) is fast to compute. For example, one may take many of them equal to 0, some equal to ± 1, several equal to the same constant a, etc. But such over-simplification can lead to bad generators (L'Ecuyer 1997, L'Ecuyer and Simard 1999, L'Ecuyer 2012, L'Ecuyer and Simard 2014). For example, the lagged-Fibonacci RNG, and its variants the add-with-carry and subtract-with-borrow generators have only two nonzero coefficients, say a_r and a_k, both equal to ± 1. As a result, their output vectors the form (u_i, u_{i-r}, u_{i-k}) all lie in only two parallel planes in the unit cube $[0,1)^3$ (Couture and L'Ecuyer 1994, L'Ecuyer 1997). These generators fail several statistical tests (L'Ecuyer and Simard 2007) and give totally incorrect results in certain types of simulations (Couture and L'Ecuyer 1994).

Combining two (or more) MRGs with the same order k but different prime moduli, say m_1 and m_2, by adding the outputs modulo 1, gives another MRG whose recurrence is modulo $m = m_1 m_2$; see L'Ecuyer (1996a). Since m is not prime, the period cannot be $m^k - 1$, but it can reach $(m^k - 1)/2$. The spectral test can be applied as usual. This combination technique provides a way of constructing fast and good quality RNGs by selecting components that allow a fast implementation and for which the recurrence of the combined MRG has good uniformity properties. L'Ecuyer (1988), L'Ecuyer and Andres (1997), L'Ecuyer (1999a), L'Ecuyer and Touzin (2000), L'Ecuyer and Simard (2007) offer several specific constructions of this type. They include `MRG32k3a` and `MRG31k3p`, which are Crush-resistant and come with multiple streams and substreams (L'Ecuyer 1999a, L'Ecuyer 2008, L'Ecuyer et al. 2002, L'Ecuyer, Munger, and Kemerchou 2015b).

3.4 Nonlinear RNGs

A nonlinear RNG can be obtained conveniently by combining two linear RNGs of different types, such as an MRG with large modulus with an $\mathbb{F}_2$-linear RNG. For this specific type of combination, L'Ecuyer and Granger-Piché (2003) derive theoretical results that provide bounds on the uniformity of Ψ_s. For the other types of nonlinear RNGs that are currently available, uniformity measures in more than one dimension are not available and appear hard to obtain. Their statistical quality can be assessed only by empirical tests.

The xorgen generators of Brent (2007) use a combination as in L'Ecuyer and Granger-Piché (2003); They combine a fast long-period $\mathbb{F}_2$-linear xorshift generator (Panneton and L'Ecuyer 2005, Brent 2007) with an additive Weyl sequence. Although the components do not have good high-dimensional uniformity, the combination is Crush-resistant. This RNG is competitive with the best $\mathbb{F}_2$-linear generators in terms of speed and period length, and does better in empirical tests.

In general, nonlinear RNGs can be constructed by taking either f or g (or both) as a nonlinear function (L'Ecuyer 1994). For example, proposals have been made in which f is a quadratic or cubic or inversive recurrence instead of a linear one (L'Ecuyer 1994, L'Ecuyer and Hellekalek 1998). They tend to be slow. Marsaglia (1985) proposed an interesting class of multiplicative lagged Fibonacci generators, which use a recurrence similar to (5), with only two nonzero coefficients a_j both equal to 1, and in which additions are replaced by multiplications. They perform very well in statistical tests (L'Ecuyer and Simard 2007).

As a general rule, either f or g must perform a sufficient amount of work to transform the state onto the next one or into the output, so that successive output values have no easily detectable dependence. For the most common RNGs, most of the transformation is done by f, while g does very little. The RNGs discussed in the next subsection do the extreme opposite: almost all the work is done by g.

3.5 Counter-Based RNGs

In *counter-based RNGs* (Hellekalek and Wegenkittl 2003, L'Ecuyer and Simard 2007, Salmon et al. 2011, Tzeng and Wei 2008, Zafar et al. 2010), f simply increments a counter: the state at step i is the counter i, so $f(i) = i + 1$. The output function g must do more work. It can be taken as a bijective block cipher encryption algorithm such as the advanced encryption standard (AES), the secure hash algorithm (SHA), the tiny encryption algorithm (TEA), MD5, ChaCha, Threefish, etc. (Claessen and Pałka 2013, L'Ecuyer and Simard 2007, Neves and Araujo 2012, Phillips et al. 2011, Salmon et al. 2011, Tzeng and Wei 2008,

Zafar et al. 2010). One advantage: it is trivial to jump ahead or backward to any given position in the sequence. These output values can then be generated in any order and are easy to replicate, It is also easy to split the sequence into long disjoint streams. When g is a bijection, the period is 2^k where k is the number of bits used to represent the counter. Typically, k is selected as a multiple of 32, for example 128 or 256. These RNGs are generally slower than popular RNGs when implemented in software, but simplified versions that are good enough for many simulation applications are practically as fast. Salmon et al. (2011) propose counter-based RNGs based on simplified (faster) versions of the AES and Threefish algorithms, named ARS and Threefry, and a new counter-based method called Philox. These RNGs were designed to be Crush-resistant. But no theoretical analysis of the uniformity of their point sets Ψ_s is available.

4 IMPLEMENTATION OF MULTIPLE STREAMS

4.1 Making and Managing Multiple Streams

The most common way to create and manage multiple streams of random numbers is by using a single central monitor, which creates the streams in a well-defined order and transfers them to the software entities that require them. This mechanism works in the same way for computations on one or many processors. Multiple streams are typically constructed by splitting a single RNG sequence into long subsequences, by computing starting points that are sufficiently far from each other. If successive streams start at ν steps from each other, to compute the starting point of the next stream from that of the current stream, one needs a function that can compute quickly $x_{i+\nu}$ from x_i. We have seen how to do that for linear recurrences.

The distance ν between streams is typically large and fixed (e.g., $\nu = 2^{127}$ in RngStreams). But users sometimes want to select a specific value of ν for a given application. This can be useful for example to ensure that a parallel version of a program gives the same results as a sequential version that uses just a single stream. Bradley et al. (2011) propose a tool that allows efficient jump-ahead for arbitrary ν, for the `MRG32k3a` RNG. Their implementation precomputes several powers of the matrix $\mathbf{A}$ mod m, which are used to compute $\mathbf{A}^\nu\mathbf{x}$ mod m for any arbitrary $\nu > 0$ and vector $\mathbf{x}$, by writing $\mathbf{A}^\nu$ mod m as a sum of powers that have been precomputed.

In some libraries such as RngStreams, clRNG, and SSJ (L'Ecuyer et al. 2002, L'Ecuyer 2008, L'Ecuyer, Munger, and Kemerchou 2015b), each stream is also divided into substreams. When comparing similar systems, each simulation run uses a new substream. After each run, all the streams are advanced to their next substream. An example of this will be given in Section 5.

Adaptations of RngStreams are proposed in various software tools such as Arena, Automod, Simul8, Inosim, SAS, Matlab, R (in the `parallel` package), etc. Examples of how to use it directly for parallel programming in OpenMP and in MPI (two popular parallel programming interfaces) are given by Karl et al. (2014).

Another approach to produce multiple streams uses a different RNG or a different set of RNG parameters for each stream (Mascagni and Srinivasan 2000, Saito and Matsumoto 2013). This is generally much less convenient than using the same RNG (and same code) for all streams, for many reasons (P. L'Ecuyer and D. Munger and B. Oreshkin and R. Simard 2015). One exception is a counter-based RNG parameterized by an encoding key in addition to the counter (which represents the state). Under the assumption that block ciphers are safe, any encoding key is as good as any other, and one can obtain "independent" streams simply by selecting a different encoding key for each stream. This assumption is not proven but seems a reasonable heuristic. Multiple streams can also be obtained easily by jumping ahead with the counter.

4.2 Multiple Streams for Vectorized Computing

The Vector Statistical Library (VSL) in the Intel Math Kernel Library (MKL) offers fast vectorized functions that exploit the single-instruction multiple-data (SIMD) facilities available in recent Intel processors (Intel Corporation 2015, Chapter 9). Eight different base RNGs are available. There are good ones and bad ones. One can have multiple streams of random numbers, but the user must provide the seed of each stream

explicitly. Functions to jump ahead by ν positions in the sequence are available for some of the RNGs. They are not available for the two MT generators `MT19937` and `SFMT19937` that are included, because jumping ahead is too slow for those, due to the huge state.

Barash and Shchur (2013) have proposed RNGSSELIB, for the same type of hardware and computing, in Fortran and C. It contains `MGR32k3a` (L'Ecuyer 1999a), `LFSR113` (L'Ecuyer 1999b), `MT19937` (Matsumoto and Nishimura 1998), and other RNGs named GL and GQ. Facilities are provided to jump ahead in the sequence, using essentially the same algorithms as in L'Ecuyer et al. (2002), L'Ecuyer (2008).

Matlab provides multiple streams of random numbers, arrays of streams, and functions to fill an array of random numbers from any stream. The streams are numbered from 1 to N for some integer N. Six type of RNGs are available, including `MRG32k3a` and a multiplicative lagged Fibonacci, which are the most recommendable for multiple streams.

4.3 Multiple Streams for GPUs

The cuRAND library (NVIDIA 2015) offers facilities for random number generation in CUDA, which is a C-type programming toolkit for NVIDIA GPUs. It provides 5 types of RNGs: `XORWOW, MRG32k3a, MTGP32, PHILOX4_32_10, MT19937`. `XORWOW` has poor quality, but the other ones are reasonably good. One can create multiple streams from each type, but the maximum number of streams is limited. The user can set the seed of any stream by passing a 64-bit integer (then the spacing between the streams is unknown) or have starting points that are regularly spaced by a fixed amount that depends on the RNG. Random numbers can be generated and consumed either on the host computer or on the GPU device.

Barash and Shchur (2014) propose PRAND, with selected RNGs implemented for both CPU and GPU usage, in Fortran, Cuda, and C with streaming SIMD extensions for improved performance. It is similar to RNGSSELIB, but also covers GPUs.

L'Ecuyer, Munger, and Kemerchou (2015a) propose clRNG, which includes various types of RNGs, all with arbitrary numbers of streams and substreams, as in RngStreams, but specially designed for the OpenCL parallel programming environment. The streams are created by a single monitor on the host computer, but the random numbers can be generated either on the host or on the device (e.g., on a GPU device). Each thread (or work item) can use several streams at the same time. The generated numbers can be integers or $\mathscr{U}(0,1)$ in `float` or in `double`. It can be found at http://simul.iro.umontreal.ca/clrng.

5 MULTIPLE STREAMS FOR COMPARING SYSTEMS: AN EXAMPLE

5.1 Multiple Streams in SSJ

In the `rng` package of SSJ (in Java) (L'Ecuyer 2008), stream objects (of type `RandomStream`) can be created in practically unlimited number, from various base RNGs. Each choice of base RNG is implemented in a different class, and a new stream is created by invoking the constructor of the class. Each stream is further partitioned into substreams. Each stream g has an initial state I_g for the stream, an initial state B_g for the current substream, and a current state C_g. At creation, one has $C_g = B_g = I_g$. This initial state is computed by the monitor at creation and is Z steps ahead of the initial state of the previously created stream. Each time a random number is generated from this stream, the current state C_g moves ahead by one position. The method `resetStartSubstream` resets C_g to B_g (it rewinds the stream to the beginning of its current substream). The method `resetNextSubstream` computes the state that is W steps ahead of B_g and resets both C_g and B_g to this new state (this is the beginning of the next substream). The method `resetStartStream` resets both C_g and B_g to I_g (it rewinds the stream to the beginning of its first substream). The values of Z and W depend on the base RNG. For `MRG32k3a`, for example, the streams have length $Z = 2^{127}$ and the substreams have length $W = 2^{76}$. To illustrate the usefulness of these tools and of multiple streams and substreams for simulation, we now give a small example taken from the documentation of SSJ.

Listing 1: Comparing two inventory policies with IRNs and CRNs

```
RandomStream streamDemand = new MRG32k3a();
RandomStream streamOrder  = new MRG32k3a();
RandomVariateGenInt genDemand = new PoissonGen (streamDemand, new PoissonDist (lambda));
Tally statDiff = new Tally ("statistics on difference");

// Simulates system for m days with the (s,S) policy; returns average profit per day.
public double simulateOneRun (int m, int s, int S, ) {
   int Xj = S, Yj;          // Stock in the morning and in the evening.
   double profit = 0.0;     // Cumulated profit.
   for (int j = 0; j < m; j++) {
      Yj = Xj - genDemand.nextInt(); // Subtract demand for the day.
      if (Yj < 0) Yj = 0;            // Lost demand.
      profit += c * (Xj - Yj) - h * Yj;
      if ((Yj < s) && (streamOrder.nextDouble() < p)) {
         // We have a successful order.
         profit -= K + k * (S - Yj);
         Xj = S;
      } else
         Xj = Yj;
   }
   return profit / m;
}

// Estimate the difference based on n runs with independent random numbers.
public void simulateDiffIRN (int n, int m, int s1, int S1, int s2, int S2) {
   statDiff.init();
   for (int i = 0; i < n; i++) {
      double value1 = simulateOneRun (m, s1, S1);
      double value2 = simulateOneRun (m, s2, S2);
      statDiff.add (value2 - value1);
   }
   statDiff.report ();
}

// Estimate the difference based on n runs with common random numbers.
public void simulateDiffCRN (int n, int m, int s1, int S1, int s2, int S2) {
   statDiff.init();
   streamDemand.resetStartStream();
   streamOrder.resetStartStream();
   for (int i = 0; i < n; i++) {
      double value1 = simulateOneRun (m, s1, S1);
      streamDemand.resetStartSubstream();
      streamOrder.resetStartSubstream();
      double value2 = simulateOneRun (m, s2, S2);
      statDiff.add (value2 - value1);
      streamDemand.resetNextSubstream();
      streamOrder.resetNextSubstream();
   }
   statDiff.report ();
}
```

5.2 An Inventory Example

We consider a simple inventory model where the demands for a given product on successive days are independent Poisson random variables with mean λ. If X_j is the stock level at the beginning of day j and D_j is the demand on that day, then there are $\min(D_j, X_j)$ sales, $\max(0, D_j - X_j)$ lost sales, and the stock at the end of the day is $Y_j = \max(0, X_j - D_j)$. Suppose there is a revenue c for each sale and a cost h for each unsold item at the end of the day. The inventory is controlled using a (s,S) policy: If $Y_j < s$, order $S - Y_j$ items, otherwise do not order. When an order is made, it arrives for the next morning with probability p, otherwise a new order must be made for the next day. When an order arrives, we pay a fixed cost K plus a marginal cost of k per item. The stock at the beginning of the first day is $X_0 = S$.

We want to estimate the expected profit for a period of m days, and compare these estimates across several values of (s,S). For this, we simulate m days, replicate this n times independently, and compute the average profit per day per replication, for each pair (s,S) of interest, It is well-known that to reduce the variance of the difference of average profits between values of (s,S) it is better to use well-synchronized common random numbers across these values (Law 2014). This means using the same random numbers to generate the demand for each day, and to decide whether each order arrives or not, across all values of (s,S).

Listing1 shows Java code that does that for two policies. The method `simulateOneRun` simulates the model for m days with a given policy (s,S) and returns the average profit. It uses two `RandomStream` objects: `streamDemand` is used to generate the demands from the Poisson distribution on successive day, and `streamOrder` is used to decide if the order arrives (with probability p) each time an order is made. To generate the demands, we construct a Poisson generator `genDemand` (this precomputes a set of tables) and use it to generate one value every day.

The method `simulateDiffIRN` simulates n runs for the two policies (s_1,S_1) and (s_2,S_2), with independent random numbers across those policies. It computes the difference of profits for each run, collect these n differences in the statistical collector `statDiff`, then estimate the expected difference in average daily profits between the two policies, computes a confidence interval on this difference, and reports the results.

The method `simulateDiffCRN` does the same, but using *common random numbers* across the two policies for each pair of simulation runs. After running the simulation with policy (s_1,S_1), the two streams are rewind to the start of their current substream, so that they produce exactly the same sequence of random numbers when the simulation is run with policy (s_2,S_2). The difference in profits is given to the statistical collector `statDiff` as before and the two streams are reset to a new substream for the next pair of simulations (the next run). By resetting the streams to new substreams, we make sure than we start at the same point for the two policies for the next run, for each stream, even if one stream has produced a different number of values (and ended at a different position) for Policy 1 than for Policy 2 in the previous run.

In this example, we generate one random number to decide if the order arrives only on the days where we make an order, and these days depend on the policy, even if the demands are the same for both policies. If we were using a single stream for both the demands and orders, a random number used to decide if the order arrives in one case could end up being used to generate a demand in the other case. This would greatly diminish the power of the CRN technology. Using two different streams as we do in Listing 1 ensures that the random numbers are used for the same purpose for the two policies.

As a numerical illustration, we tried this for $n = 500$, $m = 2000$, $\lambda = 100$, $c = 2$, $h = 0.1$, $K = 10$, $k = 1$, $p = 0.95$, $(s_1,S_1) = (80,198)$, and $(s_2,S_2) = (80,200)$. With IRNs, we obtained an average difference of 0.266 with a standard deviation of 1.530, and the 90.0% confidence interval (0.230, 0.302) for the difference. With CRNs, the corresponding values were 0.315, 0.352, and (0.307, 0.324). Using CRNs here divides the (empirical) variance of the difference by 18.85.

For this example, one may also compare $P \gg 2$ policies with CRNs. This can be done on a single processor, or with several processors in parallel, for example using each processor to perform n_2 runs for

one policy, for a total of $P * n/n_1$ processors (or threads) running in parallel. See L'Ecuyer, Munger, and Kemerchou (2015b) for a code that does that with clRNG in OpenCL.

Note that much larger variance reduction factors (VRF) than here can be obtained in other situations. In fact this factor can be arbitrarily large in some settings (L'Ecuyer and Perron 1994). For other examples and more discussion on CRNs, see L'Ecuyer and Perron (1994), L'Ecuyer and Buist (2006), L'Ecuyer (2007), Law (2014).

CRNs are commonly used for stochastic optimization via sample average optimization (Shapiro 1996, Ceẓik and L'Ecuyer 2008, Shapiro, Dentcheva, and Ruszczyński 2009). In that setting, one wishes to optimize, with respect to a (vector) parameter θ, a function defined by a mathematical expectation that depends on θ, and that one cannot compute exactly. The idea is to simulate this function n times and take the average, conceptually at every value of θ, with CRNs across all values of θ. The expectation is replaced by this sample average, which is optimized as a function of θ. In general, θ can be continuous in some region of the real space and take an infinite number of values, but during the (approximate) optimization process, the function will be evaluated only at a finite number of values of θ, with CRNs. The use of CRNs in this context is crucial.

REFERENCES

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

Barash, L. Y., and L. N. Shchur. 2013. "RNGSSELIB: Program Library for Random Number Generation: More Generators, Parallel Streams of Random Numbers, and Fortran Compatibility". *Computer Physics Communications* 184:2367–2369.

Barash, L. Y., and L. N. Shchur. 2014. "PRAND: GPU Accelerated Parallel Random Number Generation Library: Using Most Reliable Algorithms and Applying Parallelism of Modern GPUs and CPUs".

Bradley, T., J. du Toit, R. Tong, M. Giles, and P. Woodhams. 2011. "Parallelization techniques for random number generations". In *GPU Computing Gems Emerald Edition*, 231–246. Morgan Kaufmann. Chapter 16.

Brent, R. P. 2007. "Some Long-Period Random Number Generators Using Shifts and Xors". *ANZIAM Journal* 48:C188–C202.

Ceẓik, M. T., and P. L'Ecuyer. 2008. "Staffing Multiskill Call Centers via Linear Programming and Simulation". *Management Science* 54 (2): 310–323.

Claessen, K., and M. H. Pałka. 2013. "Splittable pseudorandom number generators using cryptographic hashing". In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, Haskell '13, 47–58: ACM.

Couture, R., and P. L'Ecuyer. 1994. "On the Lattice Structure of Certain Linear Congruential Sequences Related to AWC/SWB Generators". *Mathematics of Computation* 62 (206): 798–808.

Devroye, L. 1986. *Non-Uniform Random Variate Generation*. New York, NY: Springer-Verlag.

Haramoto, H., M. Matsumoto, T. Nishimura, F. Panneton, and P. L'Ecuyer. 2008. "Efficient Jump Ahead for $\mathbf{F}_2$-Linear Random Number Generators". *INFORMS Journal on Computing* 20 (3): 385–390.

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

Hörmann, W., J. Leydold, and G. Derflinger. 2004. *Automatic Nonuniform Random Variate Generation*. Berlin: Springer-Verlag.

Intel Corporation 2015. *Intel math kernel library reference manual, MKL 11.2*. Intel Corporation. Document Number 630813-065US.

Karl, A. T., R. Eubank, J. Milovanovic, M. Reiser, and D. Young. 2014. "Using RngStreams for parallel random number generation in C++ and R". *Computational Statistics* 29 (5): 1301–1320.

Khronos Group 2010. *OpenCL: the open standard for parallel programming of heterogeneous systems*. Khronos Group. See http://www.khronos.org/opencl/.

Knuth, D. E. 1998. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Third ed. Reading, MA: Addison-Wesley.

Law, A. M. 2014. *Simulation Modeling and Analysis*. Fifth ed. New York: McGraw-Hill.

L'Ecuyer, P. 1988. "Efficient and Portable Combined Random Number Generators". *Communications of the ACM* 31 (6): 742–749 and 774. See also the correspondence in the same journal, 32, 8 (1989) 1019–1024.

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. 1996a. "Combined Multiple Recursive Random Number Generators". *Operations Research* 44 (5): 816–822.

L'Ecuyer, P. 1996b. "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. 2007. "Variance Reduction's Greatest Hits". In *Proceedings of the 2007 European Simulation and Modeling Conference*, 5–12. Ghent, Belgium: EUROSIS.

L'Ecuyer, P. 2008. *SSJ: A Java Library for Stochastic Simulation*. Software user's guide, available at http://www.iro.umontreal.ca/~lecuyer.

L'Ecuyer, P. 2012. "Random Number Generation". In *Handbook of Computational Statistics* (second ed.)., edited by J. E. Gentle, W. Haerdle, and Y. Mori, 35–71. Berlin: Springer-Verlag.

L'Ecuyer, P., and T. H. Andres. 1997. "A Random Number Generator Based on the Combination of Four LCGs". *Mathematics and Computers in Simulation* 44:99–107.

L'Ecuyer, P., and E. Buist. 2006. "Variance Reduction in the Simulation of Call Centers". In *Proceedings of the 2006 Winter Simulation Conference*, 604–613: IEEE Press.

L'Ecuyer, P., and S. Côté. 1991. "Implementing a Random Number Package with Splitting Facilities". *ACM Transactions on Mathematical Software* 17 (1): 98–111.

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* 9 (2): 206–217.

L'Ecuyer, P., and J. Granger-Piché. 2003. "Combined Generators with Components from Different Families". *Mathematics and Computers in Simulation* 62:395–404.

L'Ecuyer, P., and P. Hellekalek. 1998. "Random Number Generators: Selection Criteria and Testing". In *Random and Quasi-Random Point Sets*, edited by P. Hellekalek and G. Larcher, Volume 138 of *Lecture Notes in Statistics*, 223–265. New York, NY: Springer-Verlag.

P. L'Ecuyer and D. Munger and N. Kemerchou 2015a. "clRNG: A Library for Uniform Random Number Generation in OpenCL".

P. L'Ecuyer and D. Munger and N. Kemerchou 2015b. "clRNG: A Random Number API with Multiple Streams for OpenCL". report, http://www.iro.umontreal.ca/~lecuyer/myftp/papers/clrng-api.pdf.

P. L'Ecuyer and D. Munger and B. Oreshkin and R. Simard 2015. "Random Numbers for Parallel Computers: Requirements and Methods, with Emphasis on GPUs". revision submitted, http://www.iro.umontreal.ca/~lecuyer/myftp/papers/parallel-rng-imacs.pdf.

L'Ecuyer, P., and F. Panneton. 2009. "$\mathbf{F}_2$-Linear Random Number Generators". In *Advancing the Frontiers of Simulation: A Festschrift in Honor of George Samuel Fishman*, edited by C. Alexopoulos, D. Goldsman, and J. R. Wilson, 169–193. New York: Springer-Verlag.

L'Ecuyer, P., and G. Perron. 1994. "On the Convergence Rates of IPA and FDC Derivative Estimators". *Operations Research* 42 (4): 643–656.

L'Ecuyer, P., and R. Simard. 1999. "Beware of Linear Congruential Generators with Multipliers of the Form $a = \pm 2^q \pm 2^r$". *ACM Transactions on Mathematical Software* 25 (3): 367–374.

L'Ecuyer, P., and R. Simard. 2007, August. "TestU01: A C Library for Empirical Testing of Random Number Generators". *ACM Transactions on Mathematical Software* 33 (4): Article 22.

L'Ecuyer, P., and R. Simard. 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., R. Simard, E. J. Chen, and W. D. Kelton. 2002. "An Object-Oriented Random-Number Package with Many Long Streams and Substreams". *Operations Research* 50 (6): 1073–1075.

L'Ecuyer, P., R. Simard, and S. Wegenkittl. 2002. "Sparse Serial Tests of Uniformity for Random Number Generators". *SIAM Journal on Scientific Computing* 24 (2): 652–668.

L'Ecuyer, P., and R. Touzin. 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*, 683–689. Piscataway, NJ: IEEE Press.

Marsaglia, G. 1985. "A Current View of Random Number Generators". In *Computer Science and Statistics, Sixteenth Symposium on the Interface*, edited by L. Billard, 3–10. North-Holland, Amsterdam: Elsevier Science Publishers.

Mascagni, M., and A. Srinivasan. 2000. "Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation". *ACM Transactions on Mathematical Software* 26:436–461.

Matsumoto, M., and T. Nishimura. 1998. "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator". *ACM Transactions on Modeling and Computer Simulation* 8 (1): 3–30.

Neves, S., and F. Araujo. 2012. "Fast and Small Nonlinear Pseudorandom Number Generators for Computer Simulation". In *Parallel Processing and Applied Mathematics*, Volume 7203 of *Lecture Notes in Computer Science*, 92–101. Springer.

Niederreiter, H. 1992. *Random Number Generation and Quasi-Monte Carlo Methods*, Volume 63 of *SIAM CBMS-NSF Regional Conference Series in Applied Mathematics*. Philadelphia, PA: SIAM.

NVIDIA 2015. *CURAND Library: Programming Guide, Version 7.0*. NVIDIA.

Panneton, F., and P. L'Ecuyer. 2005. "On the Xorshift Random Number Generators". *ACM Transactions on Modeling and Computer Simulation* 15 (4): 346–361.

Panneton, F., P. L'Ecuyer, and M. Matsumoto. 2006. "Improved Long-Period Generators Based on Linear Recurrences Modulo 2". *ACM Transactions on Mathematical Software* 32 (1): 1–16.

Passerat-Palmbach, J., C. Mazel, and D. R. C. Hill. 2012. "Pseudo-random streams for distributed and parallel stochastic simulations on GP-GPU". *Journal of Simulation* 6 (3): 141–151.

Phillips, C. L., J. A. Anderson, and S. C. Glotzer. 2011. "Pseudo-random number generation for Brownian Dynamics and Dissipative Particle Dynamics simulations on GPU devices". *Journal of Computational Physics* 230 (19): 7191–7201.

Saito, M., and M. Matsumoto. 2013. "Variants of Mersenne Twister Suitable for Graphic Processors". *ACM Transactions on Mathematical Software* 39 (2): 12:1–12:20.

Salmon, J. K., M. A. Moraes, R. O. Dror, and D. E. Shaw. 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*, 16:1–16:12. New York: ACM.

Shapiro, A. 1996. "Simulation-based Optimization—Convergence Analysis and Statistical Inference". *Stochastic Models* (12): 425–454.

Shapiro, A., D. Dentcheva, and A. Ruszczyński. (Eds.) 2009. *Lecture Notes on Stochastic Programming: Modeling and Theory*. Philadelphia: SIAM.

Thomas, D. B., and W. Luk. 2013. "The LUT-SR Family of Uniform Random Number Generators for FPGA Architectures". *IEEE Transactions on Very Large Scale Integration Systems* 21 (4): 761–770.

Tzeng, S., and L.-Y. Wei. 2008. "Parallel white noise generation on a GPU via cryptographic hash". In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, 79–87.

Zafar, F., M. Olano, and A. Curtis. 2010. "GPU random numbers via the tiny encryption algorithm". In *Proceedings of the Conference on High Performance Graphics*, HPG '10, 133–141.

AUTHOR BIOGRAPHY

PIERRE L'ECUYER is Professor in the Département d'Informatique et de Recherche Opérationnelle, at the Université de Montréal, Canada. He holds the Canada Research Chair in Stochastic Simulation and Optimization and an Inria International Chair in Rennes, France. He is a member of the CIRRELT and GERAD research centers. His main research interests are random number generation, quasi-Monte Carlo methods, efficiency improvement via variance reduction, sensitivity analysis and optimization of discrete-event stochastic systems, and discrete-event simulation in general. He has served as Editor-in-Chief for *ACM Transactions on Modeling and Computer Simulation* from 2010 to 2013. He is currently Associate Editor for *ACM Transactions on Mathematical Software*, *Statistics and Computing*, and *International Transactions in Operational Research*. More information can be found on his web page: http://www.iro.umontreal.ca/~lecuyer.