Proceedings of the 2021 Winter Simulation Conference
S. Kim, B. Feng, K. Smith, S. Masoud, Z. Zheng, C. Szabo, and M. Loper, eds.

MULTIPLE STREAMS WITH RECURRENCE-BASED, COUNTER-BASED, AND SPLITTABLE RANDOM NUMBER GENERATORS

Pierre L'Ecuyer

Google Research, Mountain View, CA, USA, and
DIRO, Pav. Aisenstadt, Université de Montréal
C.P. 6128, Succ. Centre-Ville
Montréal (Québec), H3C 3J7, CANADA

Olivier Nadeau-Chamard

DIRO, Pav. Aisenstadt, Université de Montréal
C.P. 6128, Succ. Centre-Ville
Montréal (Québec), H3C 3J7, CANADA

Yi-Fan Chen

Google Research
Mountain View, CA, USA

Justin Lebar

Waymo
San Francisco, CA, USA

ABSTRACT

We give an overview of the state of the art on the design and implementation of random number generators for simulation and general Monte Carlo sampling in parallel computing environments. We emphasize the need for multiple independent streams and substreams of random numbers, as well as the advantages (and potential pitfalls) of the increasingly popular counter-based and dynamically splittable generators. We look at recently-proposed constructions and software. We also recall the basic quality criteria for good random number generators and their theoretical and empirical testing. The paper outlines solutions and also raises issues that would require further study.

1 INTRODUCTION

Random number generators (RNGs) are essential ingredients for stochastic simulation, machine learning, and scientific computing in general. Their aim is to imitate the realizations of independent random variables uniformly distributed over the real interval $(0, 1)$ or the binary set $\{0, 1\}$ (L'Ecuyer 1994b; Knuth 1998; L'Ecuyer 1998; L'Ecuyer 2012). By applying appropriate transformations to these uniform random numbers, one can obtain random variates from arbitrary distributions, as well as realizations of stochastic processes and other types of random objects (Devroye 1986; Hörmann et al. 2004; Asmussen and Glynn 2007). In particular, if U has the uniform distribution over $(0, 1)$, denoted $U \sim \mathcal{U}(0, 1)$, and if F is the cumulative distribution function (cdf) of a univariate probability distribution, then $X = F^{-1}(U) = \inf\{x \in \mathbb{R} : F(x) \geq U\}$ is a random variable having the cdf F. Here we focus on how to generate realizations of U.

One natural way to obtain random numbers is by using physical devices that produce a sequence of "truly random" bits. One can take disjoint blocks of w bits to approximate $\mathcal{U}(0, 1)$ random variables U, whose binary expansion is truncated to its first w bits. Examples of such devices are thermal noise diodes, photon trajectory detectors, photon counters, and there are many others (L'Ecuyer 2017; Herrero-Collantes and Garcia-Escartin 2017; Hurley-Smith and Hernandez-Castro 2020). Some can be accessed via `/dev/random` or `/dev/urandom` on Linux distributions. While such devices are needed for cryptography, gambling machines, and loteries, they are not appropriate for applications where the reproducibility of an exact sequence of random numbers is needed, possibly even on different computers using different types of parallel architectures. This is typically the case for simulation or Monte Carlo applications, e.g., for

978-1-6654-3311-2/21/$31.00 ©2021 IEEE

verification, for comparing similar systems with common random numbers, or in optimization algorithms (Bratley et al. 1987; L'Ecuyer 2007; L'Ecuyer 2016; Law 2014; L'Ecuyer et al. 2015).

For this reason, and also because using physical devices is much less convenient than just writing code, algorithmic RNGs are highly preferred for simulation. L'Ecuyer (1990, 1994) defines an *algorithmic RNG* as a structure $(\mathcal{S}, \mu, f, \mathcal{U}, g)$ where $\mathcal{S}$ is a finite set of *states*, μ is a probability distribution on $\mathcal{S}$ to select the initial state (the *seed*) s_0, $f : \mathcal{S} \to \mathcal{S}$ is the *transition function* (usually a bijection), $\mathcal{U}$ is the *output set*, and $g : \mathcal{S} \to \mathcal{U}$ is the *output function*. At step $i \geq 1$, the *state* is $s_i = f(s_{i-1})$ and the *output* is $u_i = g(s_i) \in \mathcal{U}$. These u_i are the so-called *random numbers* produced by the RNG. We assume here that $\mathcal{U} = (0, 1)$ and we want to imitate independent $U(0, 1)$ (uniform between 0 and 1) random variables. In this definition, the function f defines a recurrence in the finite set $\mathcal{S}$. For any choice of s_0, the sequence of states s_i is necessarily periodic and the period cannot exceed $|\mathcal{S}|$, the cardinality of $\mathcal{S}$. For good RNGs, it is usually very close to that upper bound. An arbitrarily large period can be obtained easily by taking a large enough $\mathcal{S}$ and a carefully chosen f, but a larger $\mathcal{S}$ also means more memory space to store the state and more work to initialize or copy it, so a compromise must be made. Sometimes, the recurrence has two or more large cycles. Then the initial state s_0 determines which cycle is in use. For example, the MRG32k3a (L'Ecuyer 1999a) has two cycles of length near 2^{191}.

This type of *recurrence-based* generator is inherently sequential: to follow the recurrence, one needs s_{i-1} to compute s_i and u_i. This is not well-adapted to parallel computing. To launch thousands of threads that can generate random numbers in parallel on a GPU card, for example, using a single common sequence is totally ineffective. However, one can obtain *multiple streams* of random numbers by running in parallel several copies of the same recurrence with different initial states. In particular, one can use a large-period recurrence whose sequence is partitioned into long disjoint streams by selecting appropriate starting points that are spaced far apart, obtained via a jump-ahead algorithm. In many cases, there is a reasonably efficient algorithm to jump ahead directly from s_i to $s_{i+\nu}$ even for a very large ν, in particular when f is defined by a linear recurrence; see L'Ecuyer and Côté (1991), L'Ecuyer et al. (2002), Haramoto et al. (2008), Bradley et al. (2011) and the `rng` package in SSJ (L'Ecuyer and Buist 2005; L'Ecuyer 2016). But jumping ahead is usually slower than computing s_i from s_{i-1} by a factor that can go from about two (for RNGs with a reasonably small state) to a few thousands (for RNGs with a huge state, which we do not recommend). It is slower because most recurrence-based generators are designed so that f can be computed quickly, but not necessarily its ν-fold composition for a large ν. Jumping ahead repeatedly to find equally-spaced starting points is also an inherently sequential process. Doing this sequentially for tens of thousands of starting points adds significant overhead, especially if we repeat this process several times. Ideally, we want to compute all these starting points in one shot, in parallel, and not do this too often (i.e., re-use the same streams). Jumping ahead by different jump sizes to compute many starting points in parallel is doable (Bradley et al. 2011), but it requires additional storage and is more complicated than a sequential process that always uses the same jump size.

For most traditional recurrence-based RNGs, the function f transforms the state in a significant way (this makes the jumping ahead more difficult), whereas the output function g is rather simple. An opposite approach is to make f very simple and leave the complicated transformations to g. *Counter-based* RNGs (CBRNGs) push this idea to an extreme: f just increases a c-bit counter by 1 (modulo 2^c) and all the significant work is left to g. Jumping ahead then becomes trivial. In most versions, g is also parameterized by a m-bit integer k called the *key*. That is, g has two arguments: the key value k and the counter value i. One could also interpret (k, i) as a two-dimensional counter, or just a $(m + c)$-bit input. A simple way to obtain multiple streams is to have one stream for each key, with the counter starting at 0.

The CBRNG corresponds to the CTR (counter) mode of operation for block ciphers in cryptography (Dworkin 2001), which was included in particular in the *advanced encryption standard* (AES) (NIST 2001; Daemen and Rijmen 2002). Hellekalek and Wegenkittl (2003) proposed using AES in CTR mode as a robust RNG for simulation, and noted the advantages of not having to generate the random numbers in sequence: one can easily compute the output value(s) for any given pair (k, i) without having to compute

any other intermediate state. Hellekalek and Wegenkittl (2003) applied extensive statistical testing to AES in various modes and could not detect any defect. On the other hand, software implementations of AES are significantly slower than the fastest algorithmic RNGs (L'Ecuyer and Simard 2007). This motivated Salmon et al. (2011) to propose faster CBRNGs named ARS and Threefry, obtained as simplifications of the cryptographic block ciphers AES and ThreeFish, respectively, as well as a new CBRNG named Philox, designed to be fast while passing standard batteries of statistical tests from L'Ecuyer and Simard (2007). In their experiments, they compared the speeds and found that Philox was the fastest on GPUs, Threefry was the fastest on CPUs, and AES was competitive only when implemented in hardware.

Certain applications require the dynamic creation of multiple streams in a random tree-like fashion, with branches that evolve independently, without the control of a central monitor. That is, each stream must be able to split by itself in two or more children streams that evolve independently after the split, may split again, and these splits are not fully predictable. The notion of *splittable streams* discussed in Section 4 is designed to cover this situation. It builds on the concept of splittable RNG proposed by Claessen and Pałka (2013) and can provide a tree of random numbers or a tree of streams.

Multiple streams of random numbers that act as independent virtual RNGs are useful not only for parallel computing but also when running simulations on a single processor, for example to facilitate the simulation of similar systems with well-synchronized common random numbers when comparing alternative policies and for optimization (L'Ecuyer 1994a; L'Ecuyer and Buist 2006; Asmussen and Glynn 2007; Law 2014). Often, each stream is also partitioned into substreams with equally-spaced starting points, and tools are provided to jump ahead to the beginning of the next substream, jump back to the beginning of the current substream, and jump back to the beginning of the stream (L'Ecuyer et al. 2002; Karl et al. 2014; L'Ecuyer et al. 2015; L'Ecuyer 2016). This jumping can be performed independently for each stream, or for a group of streams together. As an illustration of why this is useful, suppose we want to simulate a complex service system such as a large call center, a delivery service, an emergency clinic, the ambulances in a city, a retail store, etc., where different types of demands arrive randomly, service times are random, routing decisions must be made (which agent answers this call, which nurse takes this patient, who will be served next, etc.), and we want to optimize the decision-making policy. For this, we need to simulate the system several times with each of a large number of policies that we want to consider, and make sure that for each simulation run, we use exactly the same random numbers at the same place (as much as possible) for all the policies that we simulate. This is essential to reduce the variance of the *differences* between the estimated performances across the different policies, and reduce the optimization error. For further details, see for example Ceẓik and L'Ecuyer (2008), Avramidis et al. (2010), Kleywegt et al. (2002), Shapiro et al. (2014) and the references given there. To do this properly, we would normally create many independent random streams, for example one stream to generate the arrivals for each type of "demand", one stream for each type of service time, etc. To make sure that for each simulation run, any given stream starts from the same state for all considered policies, we use one substream for each run, and we reset all streams to their next substream before each new run. We also reset all the streams to the beginning of the first substream before considering a new policy. See L'Ecuyer and Buist (2006), L'Ecuyer (2015) and the examples in (L'Ecuyer 2016) for concrete illustrations. In a different setting, in a machine learning application we may generate three types of tensors, X, Y, Z, and in a variant we may want to add a new tensor type W, or remove Y, and we want the values generated for each type to remain exactly the same, and also not depend on the order in which the types are generated. We can achieve this by assigning a different random stream to each tensor type, and making sure that each type uses the same stream in all versions.

Software facilities to manage multiple streams and substreams for recurrence-based RNGs have been proposed, e.g., in L'Ecuyer and Côté (1991), L'Ecuyer et al. (2002), L'Ecuyer and Leydold (2005), L'Ecuyer (2016), and are now widely used in stochastic simulation software designed for a single CPU, including the commercial software displayed at the WSC. Some of them have been adapted for parallel settings, including GPUs, usually with a central monitor that creates the streams (Demchik 2011; Barash and Shchur 2014; Karl et al. 2014; L'Ecuyer et al. 2015). CBRNGs provide an alternative avenue: each key gives a different

stream and the counter determine the position in the stream. They are convenient in particular when large arrays or tensors of random numbers are filled in parallel on GPUs, since finding the appropriate counter value for each compute unit is much easier than for a typical recurrence-based RNG.

We admit that there are applications for which the statistical quality of the RNG is not very important. For example, if an RNG is used just for adding noise in an iterative optimization algorithm to avoid getting stuck in local optima or to enhance exploration, it may not matter much if the random numbers are not perfectly uniform and independent. A small short-range dependence may not matter, for instance. Then, one may want to trade quality for speed. But in most simulation settings, it is important to simulate the model with the correct distributions and dependence structures. Otherwise the simulation might give wrong and misleading results, as shown in Ferrenberg et al. (1992), Tezuka et al. (1993), L'Ecuyer (1997). To avoid such situations, it is important to understand very well the structure of the RNG, ideally via a mathematical analysis. After that, empirical statistical tests can be applied (L'Ecuyer and Simard 2007).

In the remainder, we discuss and compare these various types of constructions and give examples. In Section 2, we review different ways of constructing multiple streams and substreams from recurrence-based RNGs. Section 3 covers CBRNGs. Section 4 discusses dynamically splittable RNG streams. In Section 5, we recall the main theoretical principles and quality criteria traditionally used for the design of algorithmic RNGs, and we sketch out similar principles and criteria for CBRNGs. In Section 6, we discuss empirical statistical testing and standard test suites for RNGs, and tests for multiple streams and CBRNGs. Section 7 gives a conclusion.

2 RECURRENCE-BASED RNG'S FOR PARALLEL ENVIRONMENTS

Recurrence-based RNGs with multiple streams for parallel processors have been proposed in the past and supporting software is available in many places. See L'Ecuyer (2015) and L'Ecuyer et al. (2017) for surveys. The most common design uses a single base RNG whose large period is partitioned into long streams with equally spaced starting points, say at ν steps apart. Once the initial state (seed) of the first stream is selected, the starting points of all other streams are fixed automatically. The creation of new streams is managed by a central monitor. To find the starting point of the next (newly created) stream, the monitor needs to jump ahead by ν steps in the recurrence. This is easily implemented for linear transition functions f (L'Ecuyer 1990; L'Ecuyer and Côté 1991; L'Ecuyer 1994b; Bradley et al. 2011) and for RNGs that combine different types of linear components (L'Ecuyer and Granger-Piché 2003). It also works well for nonlinear RNGs if the non-linearity is only in the output function g, for example as in the RNGs of Vigna (2016), for which f is linear modulo 2 while g uses different types of operations.

When f is linear, the jump-ahead time is roughly quadratic in k, the number of bits used to represent the state, at least for k not too large. In particular, if the state is a k-bit vector and the recurrence is linear modulo 2 ($\mathbb{F}_2$-linear), jumping ahead is achieved by multiplying the state by a $k \times k$ binary matrix, in the finite field $\mathbb{F}_2$ (L'Ecuyer and Panneton 2009). Then, one way to ensure a faster jump ahead is to use an RNG based on the combination of smaller components. For example, the LFSR113 of L'Ecuyer (1999c) combines four $\mathbb{F}_2$-linear components whose states have less than 32 bits, using a XOR for the output. Its period length is near 2^{113}. Jumping ahead then amounts to jump ahead with the smaller 32-bit components only (L'Ecuyer 2016). The MRG32k3a and MRG31k3p also combine two linear components of about the same sizes, and the jumping ahead can be done for the two components separately (L'Ecuyer et al. 2002).

For RNGs with a very large state, e.g., for the Mersenne Twister and the WELL (Matsumoto and Nishimura 1998; Panneton et al. 2006), and in other situations when jumping ahead may be too slow or too inconvenient, one alternative is to choose the starting points (or "seeds") randomly and independently instead of equally spaced. If the period is long enough and the selection is truly random, the chance of overlap may be really negligible. Specifically, if the period length is ρ and we create s streams of length ℓ with independent random starting points, the probability that there is an overlap somewhere never exceeds $s^2\ell/\rho$ (Vigna 2020). For example, if $\rho = 2^{128}$ and $s = \ell = 2^{20}$, this probability does not exceed 2^{-68}, which

is certainly negligible. This "random starting points" option can also be convenient if we do not want to rely on a central monitor to create the streams, and/or if we want to create many streams in parallel.

Unfortunately, with truly random starting points for the streams, the results are not reproducible. They can be made reproducible if we replace the truly random seeds by a sequence of seeds generated by a second RNG in a way that imitates independent random variables uniformly distributed over the set of all admissible seeds. We call this sequence a *seed schedule*. It depends on the seed of the second RNG, which we may assume is selected at random. To approximate the uniform distribution over the set of all seeds, the state space of the second RNG must be at least as large as that of the first RNG, otherwise some seeds are necessarily unreachable and the above formula for the probability of overlap no longer holds. Nevertheless, the formula still provides an upper bound if we replace ρ by the number of seeds that can be reached, assuming that they all have the same probability. For example, if the period length is 2^{128} but the user only provides a (random) 64-bit seed, and $s = \ell = 2^{20}$, we know that the probability of overlap is smaller than $s^2\ell/2^{64} = 1/16$. In implementations, the state of the second RNG is often taken much smaller than the state of the first RNG, in particular when the latter is very large and it is too bothersome for the user to provide such a large seed explicitly and store it to be able to replicate the experiment. A few successive output values from the second RNG are then taken to fill out an initial state of the first one. The probability of picking the same seed twice is much higher when we do this. If we pick s seeds at random uniformly from a pool of size r, the probability that the seeds are not all distinct is approximately $s^2/2r$ when this quantity is close to 0. This corresponds to the probability of at least one collision when throwing s points at random in r boxes (L'Ecuyer et al. 2002). For instance, if $s = 2^{30}$ and $r = 2^{64}$, this probability is near $1/32$, which is not totally negligible, but this may still be acceptable for many applications: if one billion streams are used and two of them are the same, it may be very unlikely to create visible bias for the application at hand. As an illustration, if one billion streams of random numbers are used to make the dropout decisions when training a deep neural network (Srivastava et al. 2014), and if two of these streams turn out to be the same, this is very unlikely to have a significant impact on the training.

When using a second RNG to seed the first one, we must be careful to avoid structural interaction between these two RNGs (Matsumoto et al. 2007). One may argue that it is better to use a second RNG having a different structure than the first. For example, if the first generator has an $\mathbb{F}_2$-linear transition function f, the second one should not. In the extreme situation where the two RNGs have the same transition function and the output function of the second one is the identity, the initial states of successive streams will be same as the output sequence of the first stream, which is of course very bad. If the two RNGs are too similar, bad behavior is also likely to happen. On the other hand, if they both have the same type of linear structure, it may be possible to perform a theoretical analysis of the mathematical structure of the points formed by values from different streams (see Section 5), whereas if they don't, we can only rely on empirical statistical testing for this.

Instead of having an automatic system to select and manage the seeds, as discussed so far, several recent systems leave this task to the user. Each time we want a new stream, we must provide a seed for this stream. Then the seeds are no longer random and the probabilities of overlap given earlier do not apply. When asked for 4096 different seeds, users are very unlikely to draw those seeds independently and uniformly over the set of all admissible seeds. Many will just give $\{1, 2, \ldots, 4096\}$, for example, and the system should be designed to work well even in this case. One way of handling this is to automatically apply a bijective transformation to the seeds provided by the user, e.g., with a (simplified) block cipher. This is frequently used in the context of CBRNGs (see Section 3). But even if we do this, some seeds are much more likely than others (the transformations of $1, 2, 3, \ldots$, for example)! Applying these transformations also adds overhead, which can be significant if each stream produces just a few random numbers. This option may give the user more flexibility to manage the seeds, but this flexibility comes with the responsibility of making sure that the seeds are all different and remembering them to ensure reproducibility.

As a concrete illustration, the TPUs at Google implement the `xorshift128+` generator of Vigna (2016) directly in hardware. This is an $\mathbb{F}_2$-linear recurrence-based RNG with period $2^{128} - 1$. Each TPU chip can

run 64 copies of it in parallel. Jumping ahead by ν steps with this RNG could be done by multiplying the current 128-bit state vector by a 128×128 binary matrix that depends on ν, modulo 2. But since the TPU is not adapted for this operation, the 64 initial states are obtained by transforming a single seed (only two integers) given by the user, together with a counter, to imitate 64 "random initial states." Note that one must be careful with the seed transformation. A naive idea could be to do one transition of the `xorshift128+` with initial states $1, 2, \ldots, 64$ (this is fast since the RNG is already in hardware) and combine the results in some way with the seed provided by the user, but this is not sufficient, because applying one round of the `xorshift128+` to the small integers gives numbers with many more zeros than ones. Google uses a more elaborate process to compute the 64 starting points in parallel, in terms of the given seeds.

For all these methods, the creation and assignment of streams should always be done at the logical level, independently of the hardware, to ensure reproducibility. In particular, it should *not* be one stream per processor or one stream per thread; see L'Ecuyer et al. (2017) for more discussion on this.

3 COUNTER-BASED RNG'S

Multiple streams can be trivially obtained by taking different keys (one per stream) for a CBRNG. For each stream, the counter can simply start at zero. The keys can be distributed by a central monitor, just like for recurrence-based RNGs, by using a *key schedule* (an ordered list of all possible keys), as proposed by Salmon et al. (2011). Each time a new stream is requested, the monitor picks the next key in the list. The key schedule can be specified for example by a recurrence-based RNG of period near 2^m over the set of m-bit integers, such as an LCG or an $\mathbb{F}_2$-linear recurrence as in (3), or even by another CBRNG that directly returns the ith key for each i. The latter is convenient because it can provide many keys in parallel.

Conceptually, a key is like a seed for a recurrence-based RNG. The methods described in Section 2 to select a sequence of seeds can be used to select a sequence of keys. In case we do not want a central monitor to manage a key schedule, we can think of drawing a new key at random uniformly over all 2^m possibilities, independently of the previous keys, each time a new stream must be created. If s streams are created in this way, the probability that the s keys are not all distinct is approximately $s^2/2^{m+1}$. For example, if $s = 2^{20}$ and $m = 64$, this probability is approximately 2^{-25}, whereas for $s = 2^{30}$ it is around $2^{-5} = 1/32$ according to this approximation. To ensure reproducibility, the keys should not be generated truly at random, but according to a deterministic mechanism. The selection and management of keys can also be left to the user, with the same caveats as for the seeds in Section 2, except that we do not have to worry for overlap of the streams, it suffices that the keys be distinct.

Multiple substreams (for each stream) can be defined by using a subset of the bits of either the key or the counter to determine the substream. For example, one can use the $c_0 < c$ most significant bits of the counter to determine the substream, and the remaining $c_1 = c - c_0$ bits for the position within the substream. Navigating between substreams is then very easy, provided that the user is allowed to give explicit values for the key and the counter. This can be generalized to multiple levels with multidimensional counters.

Sometimes the CBRNG returns more than one output (i.e., a vector of output values) for each value of (k, i). Then we need a secondary counter (a third argument of g) to identify the different coordinates of this vector (Salmon et al. 2011). Specifically, for each pair $(k, i) \in \{0, 1\}^{m+c}$, the CBRNG computes $\mathbf{y} = \tilde{g}(k, i)$, which is a block of $dw \le c$ bits, where d and w are positive integers. This $\mathbf{y}$ can be computed directly for any pair (k, i) without generating any other value. It is then interpreted as d blocks of w bits, and each of these blocks can be transformed into a w-bit real number in $[0, 1)$. We can use $u_{k,i,j} = g(k, i, j)$ to represent the jth output for the pair (k, i), for $j = 0, \ldots, d-1$. In most papers on CBRNGs, it is assumed that $dw = c$ and (usually) that for each value of k, $\tilde{g}(k, \cdot)$ is a bijection over the set of c-bit integers, so that each possible c-bit value of $\mathbf{y}$ appears exactly once when the counter goes from 0 to $2^c - 1$. Here, we make the definition more general to allow $d = 1$ and $w < c$, for example. When $dw < c$, we want that each value of $\mathbf{y}$ appears exactly 2^{c-dw} times. A weaker assumption would be that when the pair (k, i) runs over all its 2^{mc} possible values, each value of $\mathbf{y}$ appears exactly 2^{mc-dw} times. For good constructions,

there must be a proof that this holds. Common choices of c and m are 64, 128, and 256, for example, while common choices for w are 32 and 64.

This concept of CBRNG fits our earlier definition of algorithmic RNG if we take the pair (k, i) as the state, $f(k, i) = (k, (i+1) \mod 2^c)$, and when $d > 1$ we slightly modify the definition to view the output $g(k, i)$ as the d-dimensional vector of $U(0, 1)$ outputs produced from state (k, i). Alternatively, we can define the state as (k, i, j), in which case we have the slightly more complicated notation $f(k, i, j) = (k, i+\mathbb{I}[j = d-1], (j+1) \mod d)$ where $\mathbb{I}$ denotes the indicator function, and $u_{k,i,j} = g(k, i, j)$ is a single $U(0, 1)$ output.

Tensorflow (2019) uses the PHILOX-4×32-10 CBRNG from Salmon et al. (2011), with $m = 64$, $c = 128$, $d = 4$, and $w = 32$. It is implemented as a C++ class named `PiloxRandom`, with a low-level function that takes a pair (key, counter) as input, returns an array of four 32-bit unsigned integers, and advances the counter by 1. At a higher level, Tensorflow offers two different types of functions that use `PiloxRandom` to fill a tensor with random numbers: the *stateless* functions take an explicit seed as input and do not assume that the RNG has a "state," whereas for the *stateful* functions, each stream may have a "state" that corresponds to the value of its counter.

When calling a stateless function in Tensorflow, the user must provide a *seed* which is a pair of 32-bit or 64-bit integers. These integers are mapped by hidden transformations to four 32-bit unsigned integers. Two of them are used for the 64-bit key and the other two for the 64 most significant bits of the counter. If n random numbers are requested by the call, the lower (least significant) part of the counter goes from 0 to $n - 1$, and the n corresponding outputs are returned. Each time we call the function with the same seed, it returns the same values. Note that the seed given by the user is not directly used as a key, so if the user gives for example $(0, 0), (0, 1), (0, 2), \ldots$ as seeds in successive calls, the corresponding keys will not be so simple. Even if they were, Salmon et al. (2011) claim that this is still safe, based on their statistical testing, thanks to the ten rounds of encoding used by Philox. To control the streams, ensure that successive function calls produce different values, and also that these values are reproducible, the user must manage the seeds explicitly.

The stateful functions use two seeds: a *global seed* that can be set by a separate function `set_seed` and affects all streams, and an (optional) *operational seed* given as a parameter to the function that generates the numbers. This second seed also acts as a stream identifier: the system maintains a state (the lower 64 bits of the counter) for each operational seed that is used. Whenever a given operational seed is re-used in a function call to generate random variates, the counter associated to this particular seed (or stream) continues moving ahead from its previous value. This provides a very similar setting as in the `RNGStreams` package of L'Ecuyer et al. (2002) discussed earlier, except that here the streams are identified only by numbers (the seeds) instead of being objects with names, and there are no substreams. When a "stateful" generating function is invoked with no operational seed, the system uses a default one if a global seed has been selected, and uses a random seed otherwise.

The CBRNGs considered by Salmon et al. (2011) are all inspired by cryptographic block ciphers. They typically use several rounds of a simple encoding scheme. These rounds must be performed sequentially (they cannot be done in parallel) so more rounds means a slower generator. But for simulation, machine learning, and statistical applications, one could perhaps replace these cryptographic encoding schemes by transition functions that are typically used to design recurrence-based RNGs for simulation. Fewer rounds may then be sufficient, leading to faster CBRNGs. For example, we may consider the recurrences used for $\mathbb{F}_2$-linear generators (L'Ecuyer 1999c; L'Ecuyer and Panneton 2009; Vigna 2016; Blackman and Vigna 2021), MRGs (L'Ecuyer et al. 1993; L'Ecuyer 1996a; L'Ecuyer 1999a), *multiply with carry* (MWC) generators (Couture and L'Ecuyer 1997; Goresky and Klapper 2003), and combinations of these, perhaps with fast nonlinear transformations at the input and/or output. It may also be interesting to construct CBRNGs with $d = 1$ (instead of $d = 4$ as in Philox) because it could facilitate synchronization in SIMD settings (with GPUs).

As an example of an attempt in this direction, Widynski (2020) proposes a fast CBRNG that uses four rounds of squaring, in which both the key and the counter are of type `uint64`, and the output is a `uint32`.

The proposal passes standard empirical tests from L'Ecuyer and Simard (2007), but it fails with only three rounds, and the method is highly heuristic. For example, there is no proof that for a fixed key, all 2^{32} outputs occur the same number of times when the counter goes from 0 to $2^{64} - 1$ (block ciphers do have this property). This probably depends on the key, and keys are not all equally good. If the key is 0, the output is always 0. If the key is 1, then $x = y$ is the counter, $z = x + 1$, and the first $2^{16} - 1$ outputs are all 0 when the counter starts at 0. More generaly, if the key is a small integer k, the first $\lfloor 2^{16}/k \rfloor$ outputs (approximately) are 0. Thus, it we use $1, 2, \ldots, s$ as the keys to produce s streams, as often done in practice, the results will be very bad unless the keys are "hashed" to random-looking values by another mechanism before being used. The author recognizes implicitly that there are bad keys and suggests specific forms of keys that should be safer, but with no guarantee, and this makes the method less convenient.

Hubbard (2019) proposes a CBRNG with a five-dimensional counter, for Excel, with 10^8 possible input values. The output is an unsigned 32-bit integer, which is then converted to a real number in $(0, 1)$. The construction is ad hoc. It is unclear, for example, if each possible output appears approximately $10^8/2^{32} \approx 20$ times when the input goes over all possibilities.

4 SPLITTABLE RNG'S

There are important applications for which each stream must be able to split dynamically and randomly in two or more streams during the execution, and the occurrence of these splits is not fully predictable. That is, we want a *random tree of random numbers* instead of just linear streams. This type of splitting is needed for particle simulations in high-energy physics, where each particle has its own stream and the particles can split when they collide (Halton 1989; Mascagni and Srinivasan 2000). It also occurs with particle filters in statistics (Andrieu et al. 2010), and in computer graphics with the particles replaced by rays of light, which often split when they meet a surface (Zafar et al. 2010). When the simulations run on parallel processors (e.g., on GPUs), invoking the central monitor each time a new stream must be created in the tree can be too inefficient, and the results would not be reproducible, since the order of the calls to the central monitor may change across replications. A central monitor is not acceptable for this situation. We want a mechanism to create new streams as if their starting points were drawn randomly and independently, and are fully reproducible in the sense that each time we run the simulation, even on different hardware, we obtain exactly the same results.

The algorithmic RNG framework defined earlier can be expanded to cover this requirement. Instead of having a single transition function f, we select two or more bijective transition functions $f_1, f_2, \ldots$ in a way that for a random state s_i, the next states $f_1(s_i), f_2(s_i), \ldots$ behave as if they were independent new random states, independent of s_i, and uniformly distributed over $\mathcal{S}$. Whenever we want to split a stream into d streams, we apply $f_1, \ldots, f_d$ to the current state to obtain the states of these d streams. We can identify f_1 with f, so $f_1(s_i)$ is the next state of the current stream, while $f_j(s_i)$ gives a new stream for each $j > 1$. The key issue is how to select these bijective functions f_j. They can be the transition functions of RNGs having the same state space but totally different recurrence structures, for example. In particular, they can be d different block ciphers. To simplify, one can just take $d = 2$ and split several times when more new streams are required.

Claessen and Pałka (2013) proposed a more specific design for a splittable RNG in the form of a binary tree, where each node either returns a random number or splits in two children nodes, but not both. Conceptually, each node of the tree is identified by a string of ℓ bits if the node is at level ℓ; a 0 means we go left and a 1 means we go right in the tree, at each level. The nodes that return a random number must be leaves in the tree (they have no descendant). Storing explicitly the bit string that leads to each node would require too much memory and overhead, so to make it practical and efficient, the authors use an *iterated hashing* scheme, known as a Merkle-Damgøard construction, which compresses the representations as follows. The bit string that identifies each node is cut out in blocks of b bits. The authors take $b = 64$. A *compression (hash) function* $f : \{0, 1\}^{2b} \to \{0, 1\}^b$ is selected, as well as an initial b-bit seed (initial key) h_0. We start at the beginning of the first b-bit block m_0. Whenever a split occurs, we make two copies

of the current block m_i and add one bit: a 0 for the left branch and a 1 for the right branch. This give two new nodes with blocks m'_i and m''_i. If these new blocks now have b bits, we apply the compression function to each of them to compute the new b-bit keys $h'_{i+1} = f(h_i, m'_i)$ and $h''_{i+1} = f(h_i, m''_i)$, and these two paths move to their next blocks m_{i+1} which are now empty. This way, the current information (or "state") at any given node is always less than $2b$ bits.

For the compression function f, the authors use a *block cipher* that returns $f_k(x) = f(k, x)$ for a b-bit string x with a b-bit key k. That is, h_i is used as the encoding key in the proposed scheme. The choice of f is important for the quality of the scheme. Their implementation uses a 256-bit version of the Threefish block cipher. For the *output mapping*, the same hash function is applied to the current state (h_i, m_i). This yields the b-bit block $f(h_i, m_i)$ which is then transformed to an unsigned 32-bit integer output.

The authors report no empirical test results but they say their proofs give "strong randomness guarantees under assumptions commonly made in cryptography." These guarantees come in the form of upper bounds on the discrepancy between the RNG's output and truly independent random numbers that can be observed by any computer program that runs in reasonable time. These upper bounds hold under plausible but unproven assumptions make in cryptography, which are asymptotic as $b \to \infty$. That is, the randomness is good if b is large enough, but it is unclear what size is large enough. In particular, there is no proof that $b = 64$ is large enough for the RNG to pass all reasonable statistical tests. So in the end, the method is heuristic, like most other RNGs.

A version of this method has been adopted in JAX (Bradbury et al. 2021) a high-performance system developed for machine learning. It uses the Threefry-$2 \times 32 - 20$ algorithm from Salmon et al. (2011) as a hash function and the splitting can be in more than two new streams at a time. A stream, or state, is represented by a 64-bit key. An initialization function takes an integer seed and returns an initial key (the root of the tree). The `split` function takes a key and returns two or more new keys, whereas "`rand`" (to be replaced by the name of the distribution) functions generate tensors of random numbers from a given distribution, using the given key. A counter that starts at 0 is used internally to distinguish the entries of the tensor and its values are combined with the key in the hashing function to generate the numbers. Two or more calls to `rand` with the same key will always return the same value, so we must call both `split` and `rand` each time we want to generate new random numbers. The successive splittings form an inherently sequential process, but produce a tree whose branches can evolve in parallel. This time, in contrast to the previous descriptions, we have a tree of keys instead of a tree of random numbers. With each key (at each node), we can generate a large tensor of random numbers. The splitting does bring overhead if we generate only one (or a few) random number(s) at a time and never use the same key for both `split` and `rand`, but when filling large tensors using the counter, the overhead is minimal and the RNG is essentially as fast as the underlying Threefry.

5 THEORY AND QUALITY CRITERIA FOR ALGORITHMIC RNG'S

The theoretical analysis of recurrence-based RNGs has been done traditionally by making sure that the period is long enough (this is usually the easy part) and by computing certain measures of uniformity for the vectors of successive output values produced by the RNG, and sometimes also non-successive values at specified lags, from all possible seeds. This goes as follows (L'Ecuyer 1994b; L'Ecuyer 2012). Select s integers $0 \le i_1 < \cdots < i_s$, put $I = \{i_1, \ldots, i_s\}$, and consider the multiset

$$\Psi_I = \{\mathbf{u} = (u_{i_1}, \ldots, u_{i_s}) : s_0 \in \mathcal{S}\} \tag{1}$$

where the u_i are defined as in the introduction. It is a *multiset* in the sense that all vectors that appear more than once (if any) are counted as many times as they appear. For the special case where $I = \{0, \ldots, s-1\}$, this is the multiset of all vectors of s successive output values that can be produced by the RNG, for all possible initial states. For more general index sets I, Ψ_I contains vectors of values that are at specified lags, not necessarily successive. Ideally, we would like the vector $\mathbf{u} = (u_{i_1}, \ldots, u_{i_s})$ to have the (continuous) uniform distribution over the unit hypercube $(0, 1)^s$. But each Ψ_I is a finite multiset, and if we pick a seed

at random uniformly over all possible seeds, we get that $\mathbf{u}$ has the uniform distribution over Ψ_I. This can provide a good approximation of the ideal continuous uniform distribution only if Ψ_I covers the hypercube $(0,1)^s$ very uniformly (evenly). To be able to rigorously assess this uniformity for concrete generators, we need measures of uniformity that can be computed efficiently without generating all the points of Ψ_I explicitly (because this multiset is too large). Such computable measures exist for RNGs based on linear recurrences as outlined below (see Knuth 1998; L'Ecuyer 1996b; L'Ecuyer 1996a; L'Ecuyer 1999a; L'Ecuyer and Panneton 2009; L'Ecuyer 2012; L'Ecuyer et al. 2020) as well as for certain types of combined nonlinear generators (L'Ecuyer and Granger-Piché 2003). Of course, the measure of uniformity of Ψ_I cannot be computed for all possible index sets I, because there are too many. It can be computed for sets I of successive values up to a certain dimension, and for other sets I deemed important. For example, if we want to define multiple streams whose successive starting points in the main sequence are spaced by ν steps, it would make sense to consider sets I of the form $I = \{0, \nu, 2\nu, \dots\}$ or $I = \{0, 1, 2, \nu, \nu+1, \nu+2, 2\nu, 2\nu+1, 2\nu+2, \dots\}$, for example. The choice of index sets could also be adapted to the way the random numbers would be used in a target application. Note that for $I = \{0\}$, Ψ_I is simply $g(\mathcal{S})$, the image of $\mathcal{S}$ by g. It is usually easy to verify the uniformity of this multiset over the interval $(0,1)$.

Two important classes of RNGs for which the uniformity of the multisets Ψ_I can be measured because they have a regular mathematical structure are the linear multiple recursive generators (MRGs) and the $\mathbb{F}_2$-linear generators. MRGs are based on a linear recurrence of the form

$$x_i = (a_1 x_{i-1} + \cdots + a_k x_{i-k}) \bmod m \qquad \text{and} \qquad u_i = x_i/m, \tag{2}$$

for some coefficients $a_1, \dots, a_k$ in $\{-m+1, \dots, 0, 1, \dots, m-1\}$, with $a_k \neq 0$, k a positive integer, and m a large integer, usually a prime, in which case the period can be $m^k - 1$ by selecting the coefficients properly. Another case of interest is when $k = 1$, $m = 2^e$ for some integer e (often 32 or 64), and a constant term c is added in the recurrence, so we get

$$x_i = (a_1 x_{i-1} + c) \bmod m.$$

The period is $m = 2^e$ if and only if c is odd and $a_1 \bmod 4 = 1$ (Knuth 1998, Page 17). For an MRG, each multiset Ψ_I has a lattice structure: it is the intersection of an integral lattice with the semi-open unit hypercube $[0,1)^s$. With the constant c, the lattice is shifted, but we still have a lattice structure. This implies that all the points of Ψ_I lie in limited number of equidistant parallel hyperplanes. There are algorithms to compute the distance between the successive hyperplanes and also the minimal number of hyperplanes that contain all the points (Knuth 1998; L'Ecuyer and Couture 1997). For good uniformity, the number of hyperplanes must be large and the distance between them must be small, and this can be used to define figures of merit (Knuth 1998; L'Ecuyer 1999b). MRGs and combined MRGs selected on the basis of these types of criteria are proposed in L'Ecuyer (1999a) and L'Ecuyer and Touzin (2000), for example.

The $\mathbb{F}_2$-linear RNGs use only linear operations modulo 2. At step i, we have

$$\mathbf{x}_i = \mathbf{A}\mathbf{x}_{i-1} \bmod 2, \qquad \mathbf{y}_i = \mathbf{B}\mathbf{x}_i \bmod 2, \qquad u_i = \sum_{\ell=1}^{w} y_{i,\ell-1} 2^{-\ell}, \tag{3}$$

where $\mathbf{x}_i$ is a k-bit *state*, $\mathbf{y}_i = (y_{i,0}, \dots, y_{i,w-1})^\mathsf{T}$ a w-bit *output*, k and w are positive integers, $\mathbf{A}$ and $\mathbf{B}$ are binary matrices, and $u_i \in [0,1)$ is the returned *output*. These RNGs also have a lattice structure, but in a space of formal series (L'Ecuyer 1994b; Tezuka 1995). This implies that for certain collections of dyadic rectangular boxes of equal sizes that partition $[0,1)^s$, for any index set I, some boxes are empty and all non-empty boxes contain the same number of points from Ψ_I. When there is no empty box, which we prefer, the points are said to be *equidistributed* for this partition. For each I, equidistribution can be verified by using the lattice structure and linear algebra. A measure of uniformity can be computed based on these results for selected sets I (L'Ecuyer 1999c; Panneton et al. 2006; L'Ecuyer and Panneton 2009).

If the seeds are not equally spaced but follow a seed schedule $\tilde{s}_0, \tilde{s}_1, \tilde{s}_2, \ldots$ determined by a second RNG with state space S_2 and random initial state $\sigma_0 \in S_2$, we may consider multisets Ψ of the form

$$\Psi = \{\mathbf{u} = (g(\tilde{s}_0), g(f(\tilde{s}_0)), g(f(f(\tilde{s}_0))), \ldots, g(\tilde{s}_1), g(f(\tilde{s}_1)), g(f(f(\tilde{s}_1))), \ldots,) : \sigma_0 \in S_2\} \tag{4}$$

where we take a few successive outputs (e.g., 1 to 5) from the first seed, then a few from the second seed, etc. It may be possible to assess the mathematical structure of this set if the two RNGs have the same type of linear structure (e.g., both $\mathbb{F}_2$-linear), but otherwise it is likely to be too complicated, so one would have to rely on empirical testing only.

This type of theoretical structural analysis is also difficult to apply to CBRNGs which use highly nonlinear encoding, but it could be applied if we use some form of linear function g. To do it, we need to determine how the sets I and Ψ_I would be defined. The role of I is to select point coordinates that are likely to interact in some applications, and for which the independence matters. To simplify the notation, we assume in the following that $d = 1$, although our discussion can be easily extended to $d > 1$. One relevant choice would to define I as before and look at the uniformity of Ψ_I for one key k at a time. That is, we would replace the multiset Ψ_I by

$$\Psi_{k,I} = \{\mathbf{u} = (g(k, (i_1 + i) \bmod 2^w), \ldots, g(k, (i_s + i) \bmod 2^w)) : 0 \le i < 2^w\}. \tag{5}$$

Ideally, we would like to show that this multiset has good uniformity for each key. A second interesting choice is to construct the points by using successive keys for a fixed counter value. If $K = \{\kappa_1, \ldots, \kappa_s\}$ denotes an index set for the keys and let $\varphi(\kappa_j + \kappa)$ denote the $(\kappa_j + \kappa)$th key in the key schedule. The following set corresponds to a fixed counter value i and a random starting point κ for the key schedule:

$$\Psi_{K,i} = \{\mathbf{u} = (g(\varphi(\kappa_1 + \kappa) \bmod 2^m, i), \ldots, g(\varphi(\kappa_s + \kappa) \bmod 2^m, i)) : 0 \le \kappa < 2^m\}. \tag{6}$$

We can also consider multisets defined by interleaving the indices in various ways; for example, the first $s_1 < s$ coordinates of the points of $\Psi_{K,0}$, then the next s_1 coordinates of the points of $\Psi_{K,1}$, etc. Many other possibilities can be tested, to look for structural defects.

6 STATISTICAL TESTING

Global mathematical measures of uniformity as mentioned in the previous section cannot always be computed, and in the best case they can be computed only for a small subset of index sets I. Thus, the theoretical analysis of the structure must be supplemented by empirical statistical tests, which try to detect observable non-uniformity or lack of independence by examining output values from the generators. Various tests and batteries of tests have been designed over the last 80 years or so; see Knuth (1998), L'Ecuyer and Simard (2007), L'Ecuyer (2017) and the many references given there. Each test takes successive output values from the RNG, then computes the value taken by a test statistic T, as well as the probability that T takes a value as large (or as small) as the one it took, under the null hypothesis $\mathcal{H}_0$ that the outputs are truly uniform and independent. This probability is called the p-value, and the RNG fails the test when it is much too small (e.g., less than 10^{-10}). Good practice is to report each p-value, and not only "fail" or "pass". When a suspicious p-value is encountered, e.g., between 10^{-10} and 10^{-3}, it is recommended to run the same test again with a disjoint part of the sequence, and perhaps with a larger sample size, to see if the suspicious value was obtained just by chance (this happens) or if it indicates a real defect (if the very small p-value shows up again systematically). Usually, at least with recurrence-based RNGs, things clarify very quickly.

For a concrete example of a statistical test, recall that the hypothesis $\mathcal{H}_0$ that we want to test is equivalent to saying that for any integer $t > 0$, any tuple of t successive output values has the uniform distribution over $(0, 1)^t$. A very natural way to test this is to partition this unit cube into $k = d^t$ small subcubes by partitioning each axis $(0, 1)$ in d intervals of length $1/d$, then generate n disjoint t-dimensional tuples from the generator and count how many fall in each subcube, say X_j in subcube j for $j = 0, \ldots, k - 1$. Under

$\mathcal{H}_0$, the expected number falling in any given subcube is n/k, and if n/k is not too small, the chi-square test statistic $T = \sum_{j=0}^{k-1}(X_j - n/k)^2/(n/k)$ follows approximately the chi-square distribution with $k-1$ degrees of freedom, and can be used for the test. This is a *serial test*. To get more bang for the buck, i.e., more points for the same number of outputs, one can use overlapping t-tuples: the next t-tuple is obtained by shifting the coordinates of the current one by 1 and adding one fresh coordinate. Then T no longer has a chi-square distribution, but another related test statistic can be used (Good 1953). One drawback of these tests is that one should have $n \geq k$ (at least) for the approximation to be good, whereas we may want a larger k to detect "finer" departures from uniformity, and this implies larger values of k. To resolve this, the *collision test* uses $k \gg n$ and computes the number of times a point falls in a subcube that was already visited (the number C of collisions) instead of T. This can also be done with overlapping. Other test statistics can also be defined as functions of these X_j (L'Ecuyer et al. 2002).

Several types of statistical tests for RNGs, as well as selected batteries of tests, have been proposed and implemented. They can be classified (mostly) in two categories: those testing RNGs that imitate independent $U(0,1)$ random numbers, and those designed to test binary sequences that are supposed to imitate independent random bits. The most popular and exhaustive test suites nowadays can be found in TestU01 (L'Ecuyer and Simard 2007): the Crush batteries for $U(0,1)$ random numbers, and Rabbit and Alphabit for binary sequences. RNGs that pass the Crush batteries are often qualified as *Crush-resistant* (Salmon et al. 2011). Note that the current Crush batteries only look at the 30 most significant bits of the $U(0,1)$ outputs. A new 64-bit version of TestU01 (under development) will permit one to test up to 64 bits per integer output and 53 bits per floating-point output. As mentioned earlier, the Philox counter-based RNG and the `xorshift128+` used in Tensorflow passed these tests successfully. However, the `xorshift128+` fails some tests that focus only on the least-significant bits (Lemire and O'Neill 2019), so one should be careful not to use these bits in a "significant" way.

Statistical tests for RNGs have been traditionally designed for a single output sequence. For RNGs with multiple streams and substreams (including splittable, counter-based, etc.), it is also important to test the dependence between those streams. For that, one can construct sequences that take a few values from each stream for a certain number of streams, in a round-robin fashion, as for the theoretical analysis in Section 5. For example, one can use successive values as defined in (4) for a fixed number of streams (a power of 2 from 16 to 1024, for example), take a fixed number of values per stream (from 1 to 8, for example), and return to the first stream after the last one. For counter-based RNGs, one can define a sequence in the same way by taking a few counter values par key, or just fixing the key and advancing the counter. Repeating the latter with various keys may permit one to detect bad keys. See Section 2.2.1 of Salmon et al. (2011) for other relevant ideas. In brief, batteries of tests for counter-based RNGs need to be further developed.

For splittable RNGs, we know how to compute the probability of no collision (no key appears twice in the splitting process) when all new keys are independent random integers from the uniform distribution over the key space. But in practice, the sequence of keys is often a deterministic function of a small seed. It would then be relevant to test for the absence of collisions for "popular" seeds such as 123, for example.

Empirical testing can never *prove* that an RNG is foolproof. It is somewhat like hunting moose in a wood: if you get no moose after a day, it does not prove that there is no moose in the wood. Same thing if you get no moose after waiting 100 days at the same spot: you may be at the wrong spot! Likewise, if your favorite RNG passes a given test battery, this does not prove that it passes all tests. It it passes a set of tests with huge sample sizes, that takes days to run, it may still fail a different test that runs in a few minutes. Hence the relevance of theoretical tests. But empirical testing rightfully increases confidence and is also necessary.

7 CONCLUSION

We gave an overview of popular approaches to obtain multiple streams of random numbers for Monte Carlo methods: partitioning the sequence of a large-period recurrence-based RNG into segments of equal lengths, generating random starting points, using a counter-based RNG with a different key for each stream,

and using a splittable RNG. Each one has strengths and weaknesses, and is used in some current software. Simulation software traditionally uses the first one. Current implementations of the last two use mostly block cipher encodings, often simplified, but with many rounds that must be applied sequentially. Other types of functions could lead to faster generators and this should be explored. Parallel RNGs have also been designed in the form of vectorized constructions, where the state and output are vectors of fixed size. Some run on massively parallel array processors or field programmable gate arrays. These interesting directions are not discussed here; we refer the reader to L'Ecuyer et al. (2017) and the references therein.

ACKNOWLEDGMENTS

This work has been supported by a Discovery Grant from NSERC-Canada and also Google support to the first author. The authors are grateful to François Belletti, Yunxing Dai, Peter Hawkins, Cliff Young, and Bixia Zheng, who provided very useful information and feedback that helped improving the paper.

REFERENCES

Andrieu, C., A. Doucet, and R. Holenstein. 2010. "Particle Markov Chain Monte Carlo Methods". *Journal of the Royal Statistical Society, Series B* 72:1–33.

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

Avramidis, A. N., W. Chan, M. Gendreau, P. L'Ecuyer, and O. Pisacane. 2010. "Optimizing Daily Agent Scheduling in a Multiskill Call Centers". *European Journal of Operational Research* 200(3):822–832.

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". *Computer Physics Communications* 185(4):1343–1353.

Blackman, D., and S. Vigna. 2021. "Scrambled Linear Pseudorandom Number Generators". *ACM Transactions on mathematical Software*. To appear.

Bradbury, J., R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. 2021. *JAX: composable transformations of Python+NumPy programs*. http://github.com/google/jax, accessed 9th August 2021.

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.

Bratley, P., B. L. Fox, and L. E. Schrage. 1987. *A Guide to Simulation*. Second ed. New York, NY: Springer-Verlag.

Ceşik, 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. 1997. "Distribution Properties of Multiply-with-Carry Random Number Generators". *Mathematics of Computation* 66(218):591–607.

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

Demchik, V. 2011. "Pseudo-random number generators for Monte Carlo simulations on ATI Graphics Processing Units". *Computer Physics Communications* 182(3):692–705.

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

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.

Ferrenberg, A. M., D. P. Landau, and Y. J. Wong. 1992. "Monte Carlo Simulations: Hidden Errors From "Good" Random Number Generators". *Physical Review Letters* 69(23):3382–3384.

Good, I. J. 1953. "The Serial Test for Sampling Numbers and Other Tests for Randomness". *Proceedings of the Cambridge Philosophical Society* 49:276–284.

Goresky, M., and A. Klapper. 2003. "Efficient Multiply-with-Carry Random Number Generators with Maximal Period". *ACM Transactions on Modeling and Computer Simulation* 13(4):310–321.

Halton, J. H. 1989. "Pseudo-random Trees: Multiple Independent Sequence Generators for Parallel and Branching Computations". *Journal of Computational Physics* 84:1–56.

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):290–298.

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

Herrero-Collantes, M., and J. C. Garcia-Escartin. 2017. "Quantum random number generators". *Reviews of Modern Physics* 89:015004.

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

Hubbard, D. W. 2019. "A Multi-Dimensional Counter-Based Pseudo Random Number Generator as a Standard for Monte Carlo Simulations". In *Proceedings of the 2019 Winter Simulation Conference*, edited by N. Mustafee, K.-H. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, 3064–3073: Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Hurley-Smith, D., and J. Hernandez-Castro. 2020. "Quantum Leap and Crash: Searching and Finding Bias in Quantum Random Number Generators". *ACM Transactions on Privacy and Security* 23(3):Aticle 16, 25 pages.

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.

Kleywegt, A. J., A. Shapiro, and T. Homem-de Mello. 2002. "The sample average approximation method for stochastic discrete optimization". *SIAM Journal on Optimization* 12(2):479–502.

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. 1990. "Random Numbers for Simulation". *Communications of the ACM* 33(10):85–97.

L'Ecuyer, P. 1994a. "Efficiency Improvement via Variance Reduction". In *Proceedings of the 1994 Winter Simulation Conference*, edited by J. D. Tew, M. Manivannan, D. A. Sadowski, and A. F. Seila, 122–132: Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

L'Ecuyer, P. 1994b. "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. 1998. "Uniform Random Number Generators". In *Proceedings of the 1998 Winter Simulation Conference*, edited by D. J. Medeiros, E. F. Watson, J. S. Carson, and M. S. Manivannan, 97–104: Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

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 Linear Congruential Generators of Different Sizes and Good Lattice Structure". *Mathematics of Computation* 68(225):249–260. Errata at http://www.iro.umontreal.ca/~lecuyer/myftp/papers/latrules99Errata.pdf.

L'Ecuyer, P. 1999c. "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. 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. 2015. "Random Number Generation with Multiple Streams for Sequential and Parallel Computers". In *Proceedings of the 2015 Winter Simulation Conference*, edited by L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, 31–44: Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

L'Ecuyer, P. 2016. "SSJ: Stochastic Simulation in Java". http://simul.iro.umontreal.ca/ssj/, accessed 9th August 2021.

L'Ecuyer, P. 2017. "History of Uniform Random Number Generation". In *Proceedings of the 2017 Winter Simulation Conference*, edited by W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, 202–230: Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

L'Ecuyer, P., F. Blouin, and R. Couture. 1993. "A Search for Good Multiple Recursive Random Number Generators". *ACM Transactions on Modeling and Computer Simulation* 3(2):87–98.

L'Ecuyer, P., and E. Buist. 2005. "Simulation in Java with SSJ". In *Proceedings of the 2005 Winter Simulation Conference*, 611–620. Piscataway, NJ: Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

L'Ecuyer, P., and E. Buist. 2006. "Variance Reduction in the Simulation of Call Centers". In *Proceedings of the 2006 Winter Simulation Conference*, edited by L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, 604–613: Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

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 J. Leydold. 2005. *rstream: Streams of Random Numbers for Stochastic Simulation*.

L'Ecuyer, P., D. Munger, and N. Kemerchou. 2015. "clRNG: A Random Number API with Multiple Streams for OpenCL". report, http://www.iro.umontreal.ca/~lecuyer/myftp/papers/clrng-api.pdf, accessed 9th August 2021.

L'Ecuyer, P., D. Munger, B. Oreshkin, and R. Simard. 2017. "Random Numbers for Parallel Computers: Requirements and Methods, with Emphasis on GPUs". *Mathematics and Computers in Simulation* 135:3–17.

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 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., 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*, edited by J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, 683–689: Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

L'Ecuyer, P., P. Wambergue, and E. Bourceret. 2020. "Spectral Analysis of the MIXMAX Random Number Generators". *INFORMS Journal on Computing* 32(1):135–144.

Lemire, D., and M. E. O'Neill. 2019. "Xorshift1024*, xorshift1024+, xorshift128+ and xoroshiro128+ fail statistical tests for linearity". *Journal of Computational and Applied Mathematics* 350:139–142.

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.

Matsumoto, M., I. Wada, A. Kuramoto, and H. Ashihara. 2007. "Common Defects in Initialization of Pseudorandom Number Generators". *ACM Transactions on Modeling and Computer Simulation* 17(4):Article 15.

NIST 2001. "Advanced Encryption Standard (AES)". FIPS-197, U.S. DoC/National Institute of Standards and Technology. See http://csrc.nist.gov/CryptoToolkit/tkencryption.html, accessed 9th August 2021.

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.

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: Association for Computing Machinery.

Shapiro, A., D. Dentcheva, and A. Ruszczyński. 2014. *Lecture Notes on Stochastic Programming: Modeling and Theory*. Second ed. Handbooks in Operations Research and Management Science. Philadelphia: Society for Industrial and Applied Mathematics.

Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. 2014. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". *Journal of Machine Learning Research* 15:1929–1958.

Tensorflow 2019. "Tensorflow". See https://www.tensorflow.org, accessed 9th August 2021.

Tezuka, S. 1995. *Uniform Random Numbers: Theory and Practice*. Kluwer Academic.

Tezuka, S., P. L'Ecuyer, and R. Couture. 1993. "On the Add-with-Carry and Subtract-with-Borrow Random Number Generators". *ACM Transactions of Modeling and Computer Simulation* 3(4):315–331.

Vigna, S. 2016. "An Experimental Exploration of Marsaglia's Xorshift Generators, Scrambled". *ACM Transactions on Mathematical Software* 42(4):30:1–30:23.

Vigna, S. 2020. "On the probability of overlap of random subsequences of pseudorandom number generators". *Information Processing Letters* 158:105939.

Widynski, B. 2020. "Squares: A Fast Counter-Based RNG". https://arxiv.org/abs/2004.06278, accessed 9th August 2021.

Zafar, F., M. Olano, and A. Curtis. 2010. "GPU Random Numbers via the Tiny Encryption Algorithm". In *High Performance Graphics*, edited by M. Doggett, S. Laine, and W. Hunt, 133–141. The Eurographics Association.

AUTHOR BIOGRAPHIES

PIERRE L'ECUYER is a Professor in the Département d'Informatique et de Recherche Opérationnelle (DIRO), at Université de Montréal, Canada. He is currently a visiting scholar at Google Research. He is a member of CIRRELT and GERAD research

centers. His main research interests are random number generation, quasi-Monte Carlo methods, variance reduction, sensitivity analysis and optimization, and stochastic simulation in general. He has published 285 scientific articles and developed stochastic simulation software. He has served as a referee for 163 different scientific journals. He received the Lifetime Professional Achievement Award from the INFORMS Simulation Society in 2020. His web site is http://www.iro.umontreal.ca/~lecuyer, and his email address is lecuyer@iro.umontreal.ca.

OLIVIER NADEAU-CHAMARD is a M.Sc. student in the DIRO, at Université de Montréal, Canada. He currently works on the design and analysis of random number generators for simulation and Monte Carlo methods in parallel settings. His email address is olivier.nadeau-chamard@umontreal.ca.

YI-FAN CHEN is a Senior Staff Software Engineer at Google Research, in Mountain View, California. He has been at Google since 2012 and was with Brion technologies/ASML for five years before that. He has a Ph.D. in Applied Physics from Cornell University. His interests are in developing libraries for large-scale distributed scientific computing. His email address is yifanchen@google.com.

JUSTIN LEBAR is a Staff Software Engineer at Waymo, San Francisco, California. He has a BS and a MS in computer science from Stanford University. His main interest is high-performance programming, particularly for parallel accelerators such as GPUs. His email address is jlebar@waymo.com.