SOFTWARE FOR UNIFORM RANDOM NUMBER GENERATION: DISTINGUISHING THE GOOD AND THE BAD

Pierre L'Ecuyer

Département d'Informatique et de Recherche Opérationnelle
Université de Montréal, C.P. 6128, Succ. Centre-Ville
Montréal, H3C 3J7, CANADA

ABSTRACT

The requirements, design principles, and statistical testing approaches of uniform random number generators for simulation are briefly surveyed. An object-oriented random number package where random number streams can be created at will, and with convenient tools for manipulating the streams, is presented. A version of this package is now implemented in the *Arena* and *AutoMod* simulation tools. We also test some random number generators available in popular software environments such as Microsoft's *Excel* and *Visual Basic*, SUN's *Java*, etc., by using them on two very simple simulation problems. They fail the tests by a wide margin.

1 WHAT ARE WE LOOKING FOR?

1.1 Introduction

The aim of (pseudo)random number generators (RNGs) is to implement an imitation of the abstract mathematical concept of mutually independent random variables uniformly distributed over the interval $[0, 1]$ (i.i.d. $U[0, 1]$, for short). Such RNGs are required not only for stochastic simulation, but for many other applications involving computers, such as statistical experiments, numerical analysis, probabilistic algorithms, computer games, cryptology and security protocols in communications, gambling machines, virtual casinos over the internet, and so on. Random variables from other distributions than the standard uniform are simulated by applying appropriate transformations to the uniform random numbers (Law and Kelton 2000).

Various RNGs are available in computer software libraries. These RNGs are in fact small computer programs implementing (ideally) carefully crafted algorithms, whose design should be based on solid mathematical analysis. Are these RNGs all reliable? Unfortunately, despite repeated warnings over the past years about certain classes of generators, and despite the availability of much better alternatives, simplistic and unsafe RNGs still abound in commercial software. Concrete examples are given in Section 4 of this paper.

A single RNG does not always suffice for simulation. In many applications, several "independent" random number *streams* (which can be interpreted as distinct RNGs) are required, with appropriate tools to jump around within these streams, for instance to make independent runs and to facilitate the implementation of certain variance reduction techniques (Bratley, Fox, and Schrage 1987; Law and Kelton 2000). Packages implementing such RNG streams are now available. One of them, which we describe in Section 5, has been implemented in the most recent releases of the *Arena* and *AutoMod* simulation environments.

In the remainder of this section, we give a mathematical definition of an RNG, then we discuss design principles, quality criteria, and statistical testing. In Section 2, we review a few important classes of RNGs based on linear recurrences in modular arithmetic. In Section 3, we describe two very simple simulation problems which can be used as statistical tests (because a very good approximation of the exact answer is known). In Section 4, we see how certain widely-used generators perform on these tests. Section 5 gives a quick overview of an object-oriented RNG package with multiple streams and substreams. It offer facilities that should be included, we believe, in every serious general-purpose discrete-event stochastic simulation software. Implementations are available in C, C++, and Java.

1.2 Definition

Mathematically, an uniform RNG can be defined (see L'Ecuyer 1994) as a structure (S, μ, f, U, g), where S is a finite set of *states*, μ is a probability distribution on S used to select the *initial state* s_0 (called the *seed*), $f : S \to S$ is the *transition function*, $U = [0, 1]$ is the *output set*, and $g : S \to U$ is the *output function*.

The state evolves according to the recurrence $s_i = f(s_{i-1})$, for $i \geq 1$, and the *output* at step i is $u_i =$

$g(s_i) \in U$. These u_i are the so-called *random numbers* produced by the RNG. Because S is finite, the generator will eventually return to a state already visited (i.e., $s_{l+j} = s_l$ for some $l \geq 0$ and $j > 0$). Then, $s_{i+j} = s_i$ and $u_{i+j} = u_i$ for all $i \geq l$. The smallest $j > 0$ for which this happens is called the *period length* ρ. It cannot exceed the cardinality of S. In particular, if b bits are used to represent the state, then $\rho \leq 2^b$. Good RNGs are designed so that their period length is close to that upper bound.

Formally, this deterministic construction certainly disagrees with the concept of i.i.d. $U[0,1]$ random variables. But from a practical viewpoint, this is very convenient and experience indicates that this works fine.

1.3 Design Principles and Measures of Uniformity

How should RNGs be constructed? One obvious requirement is that the period length must be *guaranteed* to be extremely long, to make sure that no wrap-around over the cycle can occur in practice. The RNG must also be *efficient* (run fast and use little memory), *repeatable* (the ability of repeating exactly the same sequence of numbers is a major advantage of RNGs over physical devices, e.g., for program verification and variance reduction in simulation (see Law and Kelton 2000), and *portable* (i.e., work the same way in different software/hardware environments). The availability of efficient methods for jumping ahead in the sequence by a large number of steps, i.e., to quickly compute the state $s_{i+\nu}$ for any large ν, given the current state s_i, is also an important asset. It permits one to partition the sequence into long disjoint streams and substreams and to construct an arbitrary number of *virtual generators* from a single backbone RNG (see Section 5).

These requirements do not suffice. For example, the RNG defined by $s_{i+1} = s_i + 1$ if $s_i < 2^{500} - 1$, $s_{i+1} = 0$ otherwise, and $u_i = s_i/2^{500}$, satisfies them but is certainly not to be recommended.

We must remember that our goal is to imitate independent uniform random variables. That is, the successive values u_i should appear *uniform* and *independent*. They should behave (in appearance) as if the null hypothesis $\mathcal{H}_0$: "The u_i are i.i.d. $U[0,1]$" was true. This hypothesis is equivalent to saying that for each integer $t > 0$, the vector $(u_0, \ldots, u_{t-1})$ is uniformly distributed over the t-dimensional unit cube $[0,1]^t$. Clearly, $\mathcal{H}_0$ cannot be formally true, because these vectors always take their values only from the finite set

$$\Psi_t = \{(u_0, \ldots, u_{t-1}) : s_0 \in S\},$$

whose cardinality cannot exceed that of S. If s_0 is random, Ψ_t can be viewed as the *sample space* from which vectors of successive output values are taken randomly. When several t-dimensional vectors are produced by an RNG by taking non-overlapping blocks of t output values, this can be viewed in a way as picking points at random from Ψ_t, without replacement.

The idea then is to require that Ψ_t be very evenly distributed over the unit cube, so that $\mathcal{H}_0$ be *approximately true for practical purposes*, at least for moderate values of t. This suggests that the cardinality of S must be huge, to make sure that Ψ_t can fill up the unit hypercube densely enough. This is in fact a more important reason for having a large state space than just the fear of wrapping around the cycle because of too short a period length.

How do we measure the uniformity of Ψ_t? We need computable and convenient *figures of merit* that measure the evenness of its distribution. In practice, these figures of merit are often defined as measures of the *discrepancy* between the empirical distribution of the point set Ψ_t and the uniform distribution over $[0,1]^t$ (Niederreiter 1992; Hellekalek and Larcher 1998). There are several ways to define the discrepancy. This is closely related to goodness-of-fit test statistics for testing the hypothesis that a certain sample of t-dimensional points comes from the uniform distribution over $[0,1]^t$. An important criterion in choosing a specific measure is the ability to compute it efficiently without generating the points explicitly (we must keep in mind that Ψ_t is usually too large to be enumerated), and this depends on the mathematical structure of Ψ_t. For this reason, different figures of merit (i.e., different measures of discrepancy) are used in practice for analyzing different classes of RNGs. The selected figure of merit is usually computed in dimensions t up to some arbitrary integer t_1 chosen in advance. Examples of practical figures of merit are given in Section 2.2.

One may also examine certain sets of vectors of *non-successive* output values of the RNG. That is, for a fixed set of non-negative integers $I = \{i_1, i_2, \cdots, i_t\}$, measure the uniformity of the t-dimensional point set

$$\Psi_t(I) \quad = \quad \{(u_{i_1}, \ldots, u_{i_t}) \mid s_0 \in S\}, \qquad (1)$$

and do this for different choices of I. L'Ecuyer and Couture (1997) explain how to apply the spectral test in this general case. An open question is: What are the important sets I that should be considered? It is of course impossible to consider them all. As a sensible heuristic, one may consider the sets I for which t is below a certain threshold and the indices i_j's are not too far away from each other (L'Ecuyer and Lemieux 2000). One may also consider far apart indices that correspond to the starting points of disjoint streams of random numbers produced by the same underlying generator and used in parallel in a simulation.

Some would argue that Ψ_t should look like a typical set of random points over the unit cube instead of being too evenly distributed, i.e., that its structure should be chaotic, not regular. But chaotic structures are hard to analyze mathematically. It is probably safer to select RNG classes for which the structure of Ψ_t can be analyzed and understood, even if this implies more regularity, rather than selecting an RNG with a chaotic but poorly understood structure.

1.4 Empirical Statistical Testing

Once an RNG has been constructed and implemented, based hopefully on a sound mathematical analysis, it is customary and good practice to submit it to a battery of *empirical statistical tests* that try to detect empirical evidence against the hypothesis $\mathcal{H}_0$ defined previously. A test is defined by a test statistic T, function of a finite set of u_i's, and whose distribution under $\mathcal{H}_0$ is known. An infinite number of different tests can be defined. There is no universal battery of tests that can guarantee, when passed, that a given generator is fully reliable for all kinds of simulations. Passing a lot of tests may (heuristically) improve one's confidence in the RNG, but never proves that the RNG is foolproof. In fact, no RNG can pass all statistical tests. Perhaps the proper way of seeing things is that a *bad* RNG is one that fails *simple* tests, whereas a *good* RNG is one that fails only complicated tests that are very hard to find and run. This can be formalized in the framework of computational complexity, but we will not go in that direction here.

Ideally, the statistical tests should be selected in close relation with the target application, i.e., be based on a test statistic T that closely mimics the random variable of interest. But this is usually impractical, especially when designing and testing generators for general purpose software packages. For a sensitive application, it is recommended that the user tests the RNG specifically for his (or her) problem, or tries RNGs from totally different classes and compares the results.

Specific tests for RNGs are proposed by Knuth (1998), Hellekalek and Larcher (1998), Marsaglia (1985), Mascagni and Srinivasan (2000), Soto (1999), and other references given there. Experience shows that RNGs with very long periods, good structure of their set Ψ_t, and based on recurrences that are not too simplistic, pass most reasonable tests, whereas RNGs with short periods or bad structures are usually easy to crack by standard statistical tests.

2 SOME POPULAR FAMILIES OF RNG'S

2.1 Generators Based on Linear Recurrences

The most widely used RNGs by far are based on linear recurrences of the form

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

where the *modulus* m and the *order* k of the recurrence are positive integers, the *coefficients* a_l belong to $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$, and the *state* at step i is $s_i = (x_{i-k+1}, \ldots, x_i)$. If m is a prime number and if the a_l's satisfy certain conditions, the sequence $\{x_i, i \geq 0\}$ has the maximal period length $\rho = m^k - 1$ (Knuth 1998).

One way of defining the output function, in the case where m is large, is simply to take

$$u_i \;=\; x_i/m. \qquad (3)$$

The resulting RNG is known under the name of *multiple recursive generator* (MRG). When $k = 1$, we obtain the classical *linear congruential generator* (LCG). Implementation techniques for LCGs and MRGs are discussed by L'Ecuyer and Côté (1991), L'Ecuyer (1999a), L'Ecuyer and Simard (1999), L'Ecuyer and Touzin (2000), and the references given there.

Another approach is to take a small value of m, say $m = 2$, and construct each output value u_n from L consecutive x_j's by

$$u_i = \sum_{j=1}^{L} x_{is+j-1} m^{-j}, \qquad (4)$$

where s and $L \leq k$ are positive integers. If (2) has period length ρ and $\gcd(\rho, s) = 1$, (4) has period length ρ as well. For $m = 2$, u_i is thus constructed from L successive bits of the binary sequence (2), with a spacing of $s - L$ bits between the blocks of bits used to construct u_i and u_{i+1}. The resulting RNG is called a *linear feedback shift register* (LFSR) or *Tausworthe* generator (Tausworthe 1965; Niederreiter 1992). Its implementation is discussed by Fishman (1996), L'Ecuyer (1996b), L'Ecuyer and Panneton (2000), and Tezuka (1995). Important variants of the LFSR are the generalized feedback shift register (GFSR) generator and the twisted GFSR (Fushimi and Tezuka 1983; Tezuka 1995; Matsumoto and Kurita 1994; Matsumoto and Nishimura 1998; Nishimura 2000). The latter provides a very fast implementation of a huge-period generator with good structure.

Combined MRGs can be constructed by running two or more MRGs in parallel and adding their outputs modulo 1 (L'Ecuyer 1996a). This gives just another

MRG with large modulus (equal to the product of the individual moduli) and large period. Similarly, combining several LFSR generators by adding their outputs bitwise modulo 2 (i.e., by a bitwise exclusive or) yields another LFSR generator whose period length can reach the product of the periods of its components, if the latter are pairwise relatively prime (L'Ecuyer 1996b; L'Ecuyer 1999c). GFSR and twisted GFSR generators can be combined in a similar way. In both cases, combination can be seen as an efficient way of implementing RNGs with huge period lengths. These RNGs are usually designed so that their individual components have a fast implementation, whereas the combination has a complicated recurrence and excellent structural properties in the sense that its point set Ψ_t is well distributed over the unit hypercube $[0,1]^t$ for moderate values of t.

Other types of generators used in practice include the *lagged-Fibonacci* generators, the *add-with-carry*, *subtract-with-borrow* and *multiply-with-carry* generators, and several classes of *nonlinear* generators. The latter introduce nonlinearities either in the transition function f or in the output function g. Nonlinear generators are generally slower than the linear ones for a comparable period length, but some of them tend to behave better (empirically) in statistical tests because of the less regular structure of their point sets Ψ_t. For more details about these different types of generators, see, e.g., (Eichenauer-Herrmann 1995; Eichenauer-Herrmann, Herrmann, and Wegenkittl 1997; Hellekalek 1998; Knuth 1998; L'Ecuyer 1994; L'Ecuyer 1998; Lagarias 1993).

2.2 Practical Figures of Merit

For the MRG (2)–(3), it is well known that $\Psi_t = L_t \cap [0,1)^t$, where L_t is a *lattice* in the t-dimensional real space, in the sense that it can be written as the set of all integer linear combinations of t independent vectors in $\mathbb{R}^t$. This implies that Ψ_t lies on a limited number of equidistant parallel hyperplanes, at a distance (say) d_t apart (Knuth 1998). For Ψ_t to be evenly distributed over $[0,1)^t$, we want that distance d_t to be small. This d_t turns out to be equal to the inverse of the (Euclidean) length of a shortest nonzero vector in the dual lattice L_t^* to L_t, and computing d_t is called the *spectral test* (Dieter 1975; Knuth 1998; L'Ecuyer and Couture 1997).

To define a figure of merit for MRGs, one can choose an integer $t_1 > k$ (arbitrarily) and put

$$M_{t_1} = \min_{t \le t_1} d_t^* / d_t,$$

where d_t^* is an absolute lower bound on d_t given the cardinality of Ψ_t (Fishman 1996; L'Ecuyer 1999b). This figure of merit is between 0 and 1 and we seek a value close to 1 if possible. L'Ecuyer (1999a) provides tables of combined MRGs selected via this figure of merit, together with computer implementations. For non-successive indices, the set $\Psi_t(I)$ defined in (1) also has a lattice structure to which the spectral test can be applied in the same way as for successive indices (L'Ecuyer and Couture 1997; Entacher 1998).

For generators based on linear recurrences modulo 2, such as LFSR and twisted GFSR generators, we do not have the same kind of lattice structure so different figures of merit must be used. In this case, the cardinality of Ψ_t is 2^k. Suppose that we consider the ℓ most significant bits of t successive output values $u_i, \ldots, u_{i+t-1}$ from the generator. There are $2^{t\ell}$ possibilities for these bits. If each of these possibilities occurs exactly $2^{k-t\ell}$ times in Ψ_t, for all ℓ and t such that $t\ell \le k$, the RNG is called *maximally equidistributed* (ME) or *asymptotically random* (Tootill, Robinson, and Eagle 1973; L'Ecuyer 1996b). Explicit implementations of ME or nearly ME generators are given by L'Ecuyer (1999c) and Tezuka (1995). A property closely related to ME is that of a (t, m, s)-net, where one requires equidistribution for a more general class of partitions of $[0,1)^t$ into rectangular boxes, not only cubic boxes. See Niederreiter (1992) and Owen (1998) for details and references.

3 TWO SIMPLE TEST PROBLEMS

In this section, we describe two simple simulation problems which we turn into statistical tests. This can be done because we know in advance the answer to these problems. The corresponding statistical tests are not new: They are the *collision test*, studied by Knuth (1998) and L'Ecuyer, Simard, and Wegenkittl (2001), and the *birthday spacings test*, discussed by Marsaglia (1985), Knuth (1998) and L'Ecuyer and Simard (2001).

We start by cutting the interval $[0,1)$ into d equal segments, for some positive integer d. This partitions $[0,1)^t$ into $k = d^t$ cubic boxes. We then generate n points in $[0,1)^t$, independently. We define the random variable C as the number of times a point falls in a box that already had a point in it. This random variable occurs in an important practical application: It corresponds to the number of *collisions* for a perfectly uniform hashing algorithm where n keys are hashed into k memory addresses. Our first problem is to estimate $E[C]$, the mathematical expectation of C.

For our second problem, suppose that the k boxes are labeled from 0 to $k-1$, say by lexicographic order of the coordinates of their centers. Let $I_{(1)} \le I_{(2)} \le \cdots \le I_{(n)}$ be the labels of the cells that contain the points, sorted by increasing order. Define the *spacings* $S_j = I_{(j+1)} - I_{(j)}$, for $j = 1, \ldots, n-1$, and let Y be the number of values of $j \in \{1, \ldots, n-2\}$ such that $S_{(j+1)} = S_{(j)}$, where $S_{(1)}, \ldots, S_{(n-1)}$ are the spacings

sorted by increasing order. This is the number of collisions between the spacings. Here, the n points can be viewed as the birthdays of n random people in a world where years have k days, whence the name *birthday spacings* (Marsaglia 1985). Our second problem is to estimate $E[Y]$.

To simulate this model, we simply take n non-overlapping vectors of t successive output values produced by the generator. Each vector corresponds to one of the n points. In a real simulation study, we would have to repeat this scheme, say, N times, independently, then compute the sample average and variance of the N values of C and Y, and compute confidence intervals on the expectations $E[C]$ and $E[Y]$.

These expectations are actually known to a very good approximation when k is large, which we assume here. Indeed, for large k, C and Y follow approximately the Poisson distribution with means $\lambda_1 = n^2/(2k)$ and $\lambda_2 = n^3/(4k)$, respectively (Marsaglia 1985; L'Ecuyer, Simard, and Wegenkittl 2001; L'Ecuyer and Simard 2001).

In the next section, to check the robustness of certain generators for these simulation problems, we simulate just a single value of C and Y for each of several parameter sets (t, k, n), with each generator. These two random variables actually define statistical tests for the generators. We call them the *collision test* and the *birthday spacings test*, respectively. If c and y denote the values taken by these random variables in the experiment, the right p-values of the corresponding tests are

$$p^+(c) \stackrel{\text{def}}{=} P[X \geq c \mid X \sim \text{ Poisson}(\lambda_1)] \qquad (5)$$

and

$$p^+(y) \stackrel{\text{def}}{=} P[X \geq y \mid X \sim \text{ Poisson}(\lambda_2)], \qquad (6)$$

respectively. By replacing $\geq$ by $\leq$ in these definitions, one obtains the left p-values $p^-(c)$ and $p^-(y)$. If one of these p-values turns out to be extremely close to 0, this would indicate a problem with the generator (e.g., the points of Ψ_t are too regularly spread over the hypercube and this shows up when we simulate C or Y). This would mean that this generator gives a wrong answer for the corresponding simulation problem. In case of doubt, we may want to repeat the experiment several times, or with a larger sample size, to see if the suspect behavior is consistent or not.

4 HOW YOUR FAVORITE RNG FARES IN THOSE TESTS?

4.1 Some Popular Generators

We consider here the following widely-used generators.

Java. This is the generator used to implement the method `nextDouble` in the class `java.util.Random` of the Java standard library (see `http://java.sun.com/j2se/1.3/docs/api/java/util/Random.html`) It is based on a linear recurrence with period length 2^{48}, but each output value is constructed by taking two successive values from the linear recurrence, as follows:

$$\begin{aligned} x_{i+1} &= (25214903917\, x_i + 11) \mod 2^{48} \\ u_i &= (2^{27}\lfloor x_{2i}/2^{22} \rfloor + \lfloor x_{2i+1}/2^{21} \rfloor)/2^{53}. \end{aligned}$$

Note that the generator `rand48` in the Unix standard library uses exactly the same recurrence, but produces its output simply via $u_i = x_i/2^{48}$.

VB. This is the generator used in Microsoft Visual Basic (see `http://support.microsoft.com/support/kb/articles/Q231/8/47.ASP`). It is an LCG with period length 2^{24}, defined by

$$\begin{aligned} x_i &= (1140671485\, x_{i-1} + 12820163) \mod 2^{24}, \\ u_i &= x_i/2^{24}. \end{aligned}$$

Excel. This is the generator found in Microsoft Excel (see `http://support.microsoft.com/directory`). It is essentially an LCG, except that its recurrence

$$u_i = (9821.0\, u_{i-1} + 0.211327) \mod 1$$

is implemented directly for the u_i's in floating point arithmetic. Its period length actually depends on the numerical precision of the floating point numbers used for the implementation. This is not stated in the documentation and it is unclear what it is. Instead of implementing this generator in our testing package, we generated a large file of random numbers directly from Excel and feeded that file to our testing program.

LCG16807. This is the LCG defined by

$$\begin{aligned} x_i &= 16807 x_{i-1} \mod (2^{31} - 1), \\ u_i &= x_i/(2^{31} - 1), \end{aligned}$$

with period length $2^{31} - 2$, and proposed originally by Lewis, Goodman, and Miller (1969). This LCG has been widely used in many software libraries for statistics, simulation, optimization, etc., as well as in operating system libraries. It has been suggested in several books, e.g., Bratley, Fox, and Schrage (1987) and Law and Kelton (1982). Interestingly, this RNG was used in *Arena* and similar one was used in *AutoMod* (with the same modulus but with the multiplier 742938285) until recently, when the vendors of these products had the good idea to replace it with `MRG32k3a` (below). It is still used in several other simulation software products.

MRG32k3a. This is the generator proposed in Figure 1 of L'Ecuyer (1999a). It combines two MRGs of order 3 and its period length is near 2^{191}.

MT19937. This is the *Mersenne twister* generator proposed by Matsumoto and Nishimura (1998). Its period length is huge: $2^{19937} - 1$.

4.2 Results of the Collision Test

Table 1 gives the results of the collision test (our first simulation problem) applied to these generators, for $t = 2$, $d = n/16$, and n equal to different powers of 2. With this choice of d as a function of n, we have $k = n^2/256$, and the expected number of collisions is approximately $\lambda_1 = 128$ regardless of n. Note that only the $b = \log_2(d)$ most significant bits of each output value u_i is used to determine the box in which a point belongs. In the table, c represents the observed number of collisions and $p^+(c)$ or $p^-(c)$ the corresponding p-value. The value of c and the p-value are given only when the (left or right) p-value is less than 0.01. The blank entries thus correspond to non-suspicious outcomes. For the generators not given in the table, namely Java, MRG32k3a, and MT19937, none of the results were suspicious.

The VB generator clearly fails the test for all $n \geq 2^{15}$: the number of collisions is much too small. Note that the test with $n = 2^{16}$ requires only 131072 random numbers from the generator, which is much much less than its period length, and yet the left p-value is smaller than 10^{-15}, which means that it is extremely improbable to observe such a small number of collisions (38) just by chance. For $n = 2^{17}$ and more, we observed no collision at all! Actually, we ran the test for higher powers of 2 (the results are not shown in the table) and there was no collision for n up to 2^{23}, but 8388608 collisions (way too many) for $n = 2^{24}$. We also repeated the test with the VB generator by throwing away the first 10 bits of each output value and using the following bits instead, that is, replacing each u_i by $2^{10}u_i \mod 1$. The result was that there was way too many collisions, with the right p-values being smaller than 10^{-15} for all $n \geq 2^{14}$. For example, there was already 8192 collisions for $n = 2^{14}$ and 253952 collisions for $n = 2^{18}$. The explanation is that the VB generator is an LCG with power-of-2 modulus, and for these generators (in general) the least significant bits have a much smaller period length than the most significant ones (e.g., L'Ecuyer 1990).

The Excel generator starts failing for slightly larger sample sizes: The right p-value is less than 10^{-9} for $n = 2^{18}$ (approximately a quarter of a million) and less than 10^{-15} for $n = 2^{19}$ (just over half a million). In this case, there are too many collisions. LCG16807 starts to fail at $n = 2^{19}$ (half a million points). The other generators passed all these tests.

4.3 Results of the Birthday Spacings Test

For the birthday spacings problem, we first took $t = 2$ and $d^2 = n^3/4$, so that the expected number of collisions was $\lambda_2 = 1$ for all n. Again, the first $b = \log_2(d)$ bits of each u_i are used to determine the boxes where the points fall. Table 2 gives the test results.

The Java, VB, Excel, and LCG16807 generators start failing decisively with $n = 2^{18}$, 2^{10}, 2^{14}, and 2^{14}, respectively. These are quite small numbers of points. For as few as $n = 2^{14} = 16384$ points, the number of collisions was already 11129 for VB, 71 for Excel, and 150 for LCG16807. The probability that a Poisson random variable with mean 1 takes any of these values is so tiny that we cannot believe this occurred by chance.

Table 3 gives the results of three-dimensional birthday spacings tests ($t = 3$) with $d = n/2$, for which $\lambda_2 = 2$ for all n. Table 4 gives the results for the same tests, but with each u_i replaced by $2^{10}u_i \mod 1$; i.e., the first 10 bits of each u_i are thrown away and the test uses the next $\log_2(d)$ bits.

The VB generator fails very quickly in these tests, especially when we look at the less significant bits (Table 4): With as few as $n = 256$ points, we already have 52 collisions and a p-value smaller than 10^{-15}. The Excel generator starts failing decisively at $n = 2^{17}$. The Java generator passes these tests when we take its most significant bits, but starts failing at $n = 2^{15}$ points when we throw away the first 10 bits. The LCG16807 generator starts failing decisively already for $n = 2^{14}$ (some sixteen thousand points), in both Tables 3 and 4. MRG32k3a and MT19937 gave no suspect p-value.

Looking at the test results in the tables, we observe the following kind of behavior: When the sample size n is increased for a given test-RNG combination, the test starts to fail decisively when n reaches some critical value, and the failure is clear for all larger values of n. This kind of behavior is typical and was also observed for other statistical tests and other classes of generators (L'Ecuyer and Hellekalek 1998). Can we predict this critical value beforehand? What we have in mind here is a relationship of the form, say,

$$n_0 \approx K\rho^\gamma,$$

for a given type of test and a given class of generators, where ρ is the period length of the generator, K and γ are constants, and n_0 is the minimal sample size for which the generator clearly fails the test.

This has been achieved by L'Ecuyer and Hellekalek (1998), L'Ecuyer, Simard, and Wegenkittl (2001), L'Ecuyer, Cordeau, and Simard (2000), and L'Ecuyer and

Table 1: Results of the Collision Tests

n	d	VB		Excel		LCG16807	
		c	$p^-(c)$	c	$p^+(c)$	c	$p^+(c)$
2^{15}	2^{11}	75	3.1×10^{-7}				
2^{16}	2^{12}	38	$< 10^{-15}$				
2^{17}	2^{13}	0	$< 10^{-15}$	170	2.2×10^{-4}		
2^{18}	2^{14}	0	$< 10^{-15}$	202	9.5×10^{-10}		
2^{19}	2^{15}	0	$< 10^{-15}$	429	$< 10^{-15}$	195	2.2×10^{-8}
2^{20}	2^{16}	0	$< 10^{-15}$	—	—	238	$< 10^{-15}$

Table 2: Results of the Birthday Spacings Tests with $t = 2$

n	d	Java		VB		Excel		LCG16807	
		y	$p^+(y)$	y	$p^+(y)$	y	$p^+(y)$	y	$p^+(y)$
2^{10}	2^{14}			10	1.1×10^{-7}				
2^{12}	2^{17}			592	$< 10^{-15}$	5	3.7×10^{-3}		
2^{14}	2^{20}			11129	$< 10^{-15}$	71	$< 10^{-15}$	150	$< 10^{-15}$
2^{16}	2^{23}			64063	$< 10^{-15}$	558	$< 10^{-15}$	10066	$< 10^{-15}$
2^{18}	2^{26}	18	$< 10^{-15}$	261604	$< 10^{-15}$	4432	$< 10^{-15}$	183764	$< 10^{-15}$

Table 3: Results of the Birthday Spacings Tests with $t = 3$

n	d	Java		VB		Excel		LCG16807	
		y	$p^+(y)$	y	$p^+(y)$	y	$p^+(y)$	y	$p^+(y)$
2^{10}	2^{9}								
2^{11}	2^{10}			23	$< 10^{-15}$				
2^{12}	2^{11}			188	$< 10^{-15}$				
2^{13}	2^{12}			1159	$< 10^{-15}$			10	4.6×10^{-5}
2^{14}	2^{13}			5975	$< 10^{-15}$			92	$< 10^{-15}$
2^{15}	2^{14}			21025	$< 10^{-15}$			799	$< 10^{-15}$
2^{16}	2^{15}			55119	$< 10^{-15}$	9	2.4×10^{-4}	5995	$< 10^{-15}$
2^{17}	2^{16}			123181	$< 10^{-15}$	33	$< 10^{-15}$	34697	$< 10^{-15}$
2^{18}	2^{17}	8	1.1×10^{-3}	256888	$< 10^{-15}$	117	$< 10^{-15}$	139977	$< 10^{-15}$

Table 4: Results of the Birthday Spacings Tests with $t = 3$, with the First 10 Bits Thrown Away

n	d	Java		VB		Excel		LCG16807	
		y	$p^+(y)$	y	$p^+(y)$	y	$p^+(y)$	y	$p^+(y)$
2^{8}	2^{7}			52	$< 10^{-15}$				
2^{10}	2^{9}			672	$< 10^{-15}$				
2^{12}	2^{11}			3901	$< 10^{-15}$				
2^{13}	2^{12}			8102	$< 10^{-15}$			7	4.5×10^{-3}
2^{14}	2^{13}			16374	$< 10^{-15}$			96	$< 10^{-15}$
2^{15}	2^{14}	18	6.2×10^{-12}	32763	$< 10^{-15}$			736	$< 10^{-15}$
2^{16}	2^{15}	76	$< 10^{-15}$	65531	$< 10^{-15}$	7	4.5×10^{-3}	6009	$< 10^{-15}$
2^{17}	2^{16}	709	$< 10^{-15}$	—	—	34	$< 10^{-15}$	34474	$< 10^{-15}$
2^{18}	2^{17}	685	$< 10^{-15}$	—	—	186	$< 10^{-15}$	140144	$< 10^{-15}$

Simard (2001) for certain classes of RNGs and tests. For LCGs and MRGs with good spectral test behavior, for example, we have obtained the relationships $n_0 \approx 16\,\rho^{1/2}$ for the collision test and $n_0 \approx 16\,\rho^{1/3}$ for the birthday spacings test. This means that if we want our LCG or MRG to be safe with respect to these tests, we must construct it with a period length ρ large enough so that generating $\rho^{1/3}$ numbers is practically unfeasible. For example, $\rho > 2^{150}$ satisfies this requirement, but $\rho \approx 2^{32}$ or even $\rho \approx 2^{48}$ does not satisfy it. In particular, keeping an LCG with modulus $2^{31} - 1$ and changing the multiplier 16807 to another number does not cure the problem.

5 A MULTIPLE-STREAM PACKAGE

What kind of software do we need for uniform random number generation in a general-purpose discrete-event simulation environment? The availability of multiple streams of random numbers, which can be interpreted as independent RNGs from the user's viewpoint, is a must in modern simulation software (Law and Kelton 2000). Such multiple streams greatly facilitate the implementation of certain variance reduction techniques (such as common random numbers, antithetic variates, etc.) and are also useful for simulation on parallel processors.

One way of implementing such multiple streams is to compute seeds that are spaced far apart in the RNG sequence, and use the RNG subsequences starting at these seeds as if they were independent sequences (or *streams*) (Bratley, Fox, and Schrage 1987; Law and Kelton 2000; L'Ecuyer and Côté 1991; L'Ecuyer and Andres 1997). These streams are viewed as distinct independent RNGs. In some of the software available until a few years ago, N seeds were precomputed spaced Z steps apart, say, for small values of N such as (for example) $N = 10$ or $N = 32$.

In the Java class `java.util.Random`, RNG streams can be declared and constructed dynamically, without limit on their number. However, no precaution seems to have been taken regarding the independence of these streams. A package with multiple streams and which supports different types of RNGs is also proposed by Mascagni and Srinivasan (2000). Its design differs significantly from the one we will now discuss.

L'Ecuyer, Simard, Chen, and Kelton (2001) have recently constructed an object-oriented RNG package with multiple streams, where the streams are also partitioned into disjoint substreams, and where convenient tools are provided to move around within and across the streams and substreams. The backbone generator for this package is `MRG32k3a`, mentioned in Section 4. The spacings between the successive streams and substreams have been determined by applying the spectral test to the set $\Psi_t(I)$ of vectors of non-successive output values of the form $(u_n, \ldots, u_{n+s-1}, u_h, \ldots, u_{n+h+s-1}, u_{n+2h}, \ldots, u_{n+2h+s-1}, \ldots)$, for different values of h, s, and t. The spacings were chosen as large values of h for which the spectral test gave good results for all $s \leq 16$ and $t \leq 32$ (these upper bounds for s and t were chosen arbitrarily). The successive streams actually start $Z = 2^{127}$ steps apart, and each stream is partitioned into 2^{51} adjacent substreams of length $W = 2^{76}$.

Let us denote the initial state (seed) of a given stream g by I_g. If $s_0 = I_1$ is the initial seed of the generator and f its transition function, then we have $I_2 = T^Z(s_0)$, $I_3 = T^Z(I_2) = T^{2Z}(s_0)$, etc. The first *substream* of stream g starts in state I_g, the second one in state $T^W(I_g)$, the third one in state $T^{2W}(I_g)$, and so on. At any moment during a simulation, stream g is in some state, say C_g. We denote by B_g the starting state of the substream that contains the current state, i.e., the beginning of the *current substream*, and $N_g = T^W(B_g)$ the starting state of the *next substream*.

The software provides tools for creating new streams (without limit, for practical purposes), and to reset any given stream to its initial seed, or to the beginning of its current substream, or to its next substream. This kind of framework with multiple streams and substreams was already implemented in L'Ecuyer and Côté (1991) and L'Ecuyer and Andres (1997), but with a predefined number of streams and based on different (smaller) generators.

Figure 1 describes a Java version of the RNG package of L'Ecuyer, Simard, Chen, and Kelton (2001). (These authors describes a C++ version.) Implementations in C, C++, and Java, as well as test programs, are available at `http://www.iro.umontreal.ca/~lecuyer`. A C version of this package is now implemented in the most recent releases of *Arena* (release 5.0) and *AutoMod* (release 10.5) simulation environments (see `http://www.arenasimulation.com` and `http://www.autosim.com/index.asp`). The author is also working on implementations of RNG classes with the same interface (except for a few details), but based on different types of RNGs.

6 CONCLUSION

Do not trust the random number generators provided in popular commercial software such as Excel, Visual Basic, etc., for serious applications. Some of these RNGs give totally wrong answers for the two simple simulation problems considered in this paper. Much better RNG tools are now available, as we have just explained in this paper. Use them. If reliable RNGs are not avail-

```
public class RandMrg {
```

`public RandMrg()`
Constructs a new stream.

`public RandMrg (String name)`
Constructs a new stream with identifier `name`.

`public static void setPackageSeed (long seed[])`
Sets the initial seed for the class `RandMrg` to the six integers in the vector `seed[0..5]`. This will be the seed (initial state) of the first stream. By default, this seed is $(12345, 12345, 12345, 12345, 12345, 12345)$.

`public void resetStartStream ()`
Reinitializes the stream to its initial state: C_g and B_g are set to I_g.

`public void resetStartSubstream ()`
Reinitializes the stream to the beginning of its current substream: C_g is set to B_g.

`public void resetNextSubstream ()`
Reinitializes the stream to the beginning of its next substream: N_g is computed, and C_g and B_g are set to N_g.

`public void increasedPrecis (boolean incp)`
If `incp = true`, each RNG call with this stream will now give 53 bits of resolution instead of 32 bits (assuming that the machine follows the IEEE-754 floating-point standard), and will advance the state of the stream by 2 steps instead of 1.

`public void setAntithetic (boolean a)`
If `a = true`, the stream will now generate antithetic variates.

`public void writeState ()`
Prints the current state of this stream.

`public double[] getState()`
Returns the current state of this stream.

`public double randU01 ()`
Returns a $U[0, 1]$ (pseudo)random number, using this stream, after advancing its state by one step.

`public int randInt (int i, int j)`
Returns a (pseudo)random number from the discrete uniform distribution over the integers $\{i, i + 1, \ldots, j\}$, using this stream. Calls `randU01` once.

```
}
```

Figure 1: The Java Class `RandMrg`, which Provides Multiple Streams and Substreams of Random Numbers

able in your favorite software products, tell the vendors and insist that this is a very important issue. An expensive house built on shaky foundations is a shaky house. This applies to expensive simulations as well.

ACKNOWLEDGMENTS

This work has been supported by NSERC-Canada Grant No. ODGP0110050 and FCAR-Québec Grant No. 00ER3218. The author thanks Richard Simard, who ran the statistical tests and helped improving the paper. George Fishman and Steve Roberts suggested looking at the Excel and VB generators and provided pointers to their documentation. David Kelton and Jerry Banks helped convincing the vendors of Arena and AutoMod to change their generators for the better.

REFERENCES

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

Dieter, U. 1975. How to calculate shortest vectors in a lattice. *Mathematics of Computation* 29 (131): 827–833.

Eichenauer-Herrmann, J. 1995. Pseudorandom number generation by nonlinear methods. *International Statistical Reviews* 63:247–255.

Eichenauer-Herrmann, J., E. Herrmann, and S. Wegenkittl. 1997. A survey of quadratic and inversive congruential pseudorandom numbers. In *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, ed. P. Hellekalek, G. Larcher, H. Niederreiter, and P. Zinterhof, Volume 127 of *Lecture Notes in Statistics*, 66–97. New York: Springer.

Entacher, K. 1998. Bad subsequences of well-known linear congruential pseudorandom number generators. *ACM Transactions on Modeling and Computer Simulation* 8 (1): 61–70.

Fishman, G. S. 1996. *Monte Carlo: Concepts, algorithms, and applications*. Springer Series in Operations Research. New York: Springer-Verlag.

Fushimi, M., and S. Tezuka. 1983. The k-distribution of generalized feedback shift register pseudorandom numbers. *Communications of the ACM* 26 (7): 516–523.

Hellekalek, P. 1998. Good random number generators are (not so) easy to find. *Mathematics and Computers in Simulation* 46:485–505.

Hellekalek, P., and G. Larcher. (Eds.) 1998. *Random and quasi-random point sets*, Volume 138 of *Lecture Notes in Statistics*. New York: Springer.

Knuth, D. E. 1998. *The art of computer programming, volume 2: Seminumerical algorithms*. Third ed. Reading, Mass.: Addison-Wesley.

Lagarias, J. C. 1993. Pseudorandom numbers. *Statistical Science* 8 (1): 31–39.

Law, A. M., and W. D. Kelton. 1982. Confidence intervals for steady-state simulation, ii: A survey of sequential procedures. *Management Science* 28:550–562.

Law, A. M., and W. D. Kelton. 2000. *Simulation modeling and analysis*. Third ed. New York: McGraw-Hill.

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. 1998. Uniform random number generators. In *Proceedings of the 1998 Winter Simulation Conference*, 97–104: IEEE Press.

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.

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

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., J.-F. Cordeau, and R. Simard. 2000. Close-point spatial tests and their application to random number generators. *Operations Research* 48 (2): 308–317.

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 P. Hellekalek. 1998. Random number generators: Selection criteria and testing. In *Random and Quasi-Random Point Sets*, ed. P. Hellekalek and G. Larcher, Volume 138 of *Lecture Notes in Statistics*, 223–265. New York: Springer.

L'Ecuyer, P., and C. Lemieux. 2000. Variance reduction via lattice rules. *Management Science* 46 (9): 1214–1235.

L'Ecuyer, P., and F. Panneton. 2000. A new class of linear feedback shift register generators. In *Proceedings of the 2000 Winter Simulation Conference*, ed. J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, 690–696. Pistacaway, NJ: IEEE Press.

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. 2001. On the performance of birthday spacings tests for certain families of random number generators. *Mathematics and Computers in Simulation* 55 (1–3): 131–137.

L'Ecuyer, P., R. Simard, E. J. Chen, and W. D. Kelton. 2001. An object-oriented random-number package with many long streams and substreams. Submitted.

L'Ecuyer, P., R. Simard, and S. Wegenkittl. 2001. Sparse serial tests of uniformity for random number generators. *SIAM Journal on Scientific Computing*. To appear. Also GERAD report G-98-65, 1998.

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*, ed. J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, 683–689. Pistacaway, NJ: IEEE Press.

Lewis, P. A. W., A. S. Goodman, and J. M. Miller. 1969. A pseudo-random number generator for the system/360. *IBM System's Journal* 8:136–143.

Marsaglia, G. 1985. A current view of random number generators. In *Computer Science and Statistics, Sixteenth Symposium on the Interface*, 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 Y. Kurita. 1994. Twisted GFSR generators II. *ACM Transactions on Modeling and Computer Simulation* 4 (3): 254–266.

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.

Niederreiter, H. 1992. *Random number generation and quasi-Monte Carlo methods*, Volume 63 of *SIAM CBMS-NSF Regional Conference Series in Applied Mathematics*. Philadelphia: SIAM.

Nishimura, T. 2000. Tables of 64-bit Mersenne twisters. *ACM Transactions on Modeling and Computer Simulation* 10 (4): 348–357.

Owen, A. B. 1998. Latin supercube sampling for very high-dimensional simulations. *ACM Transactions of Modeling and Computer Simulation* 8 (1): 71–102.

Soto, J. 1999. Statistical testing of random number generators. Available at `http://csrc.nist.gov/rng/rng5.html`.

Tausworthe, R. C. 1965. Random numbers generated by linear recurrence modulo two. *Mathematics of Computation* 19:201–209.

Tezuka, S. 1995. *Uniform random numbers: Theory and practice*. Norwell, Mass.: Kluwer Academic Publishers.

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

AUTHOR BIOGRAPHY

PIERRE L'ECUYER is a professor teaching simulation in the "Département d'Informatique et de Recherche Opérationnelle", at the University of Montreal, Canada. He received a Ph.D. in operations research in 1983, from the University of Montréal. He obtained the prestigious *E. W. R. Steacie* Grant in 1995-97 and the *Killam* Grant in 2001. 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. His recent research articles are available on-line at `http://www.iro.umontreal.ca/~lecuyer`.