AN OBJECT-ORIENTED RANDOM-NUMBER PACKAGE WITH MANY LONG STREAMS AND SUBSTREAMS

PIERRE L'ECUYER

Département d'informatique et de recherche opérationnelle, Université de Montréal, C.P. 6128, succ. Centre-Ville, Montréal, Québec, Canada, H3C 3J7
lecuyer@iro.umontreal.ca

RICHARD SIMARD

Département d'informatique et de recherche opérationnelle, Université de Montréal, C.P. 6128, succ. Centre-Ville, Montréal, Québec, Canada, H3C 3J7
simardr@iro.umontreal.ca

E. JACK CHEN

BASF Corporation, 3000 Continental Drive-North, Mount Olive, New Jersey 07828-1234, chenej@basf.com

W. DAVID KELTON

Department of Quantitative Analysis and Operations Management, College of Business Administration, University of Cincinnati, Cincinnati, Ohio 45221-0130, david.kelton@uc.edu

(Received December 2000; revision received August 2001; accepted December 2001)

Multiple independent streams of random numbers are often required in simulation studies, for instance, to facilitate synchronization for variance-reduction purposes, and for making independent replications. A portable set of software utilities is described for uniform random-number generation. It provides for multiple generators (streams) running simultaneously, and each generator (stream) has its sequence of numbers partitioned into many long disjoint contiguous substreams. The basic underlying generator for this implementation is a combined multiple-recursive generator with period length of approximately 2^{191}, proposed by L'Ecuyer (1999a). A C++ interface is described here. Portable implementations are available in C, C++, and Java via the online companion to this paper on the *Operations Research* Web site. ⟨http://or.pubs.informs.org/pages/collect.html⟩.

Experts now recognize that small linear congruential generators (LCGs) with moduli around 2^{31} or so should no longer be used as general-purpose random-number generators (RNGs). Not only can one exhaust the period in a few minutes on a PC, but more importantly, the poor structure of the points can dramatically bias simulation results for sample sizes much smaller than the period length.

As an example, L'Ecuyer and Simard (2001) consider a simple simulation problem where n points are generated randomly in k cells over the two-dimensional square, to estimate the expected number of *repeated* values for the spacings between successive cells that contain a point. They find that with an LCG of the form

$$x_i = ax_{i-1} \bmod m, \quad u_i = x_i/m, \quad x_0 \in \{1, \ldots, m-1\},$$

if k is large ($\approx n^3/4$) and $n \approx 8m^{1/3}$ (or more), the simulation gives totally wrong results, regardless of m and a. This means only $n \approx 10,000$ for $m \approx 2^{31}$ and $n \approx 500,000$ for $m \approx 2^{48}$.

Much better RNGs have already been proposed to replace older unsafe LCGs. We mention, for instance, the Mersenne twister of Matsumoto and Nishimura (1998), the combined MRGs of L'Ecuyer (1999a), the combined LCGs of L'Ecuyer and Andres (1997), and the combined Tausworthe generators of L'Ecuyer (1999b). All of these have fairly solid theoretical support, have been extensively tested, and are easy to use.

However, a single RNG does not always suffice. Many disjoint random-number subsequences, each having long period and good statistical properties, are often required in simulation studies, for instance, to make independent replications or to associate distinct "streams" of random numbers with different sources of randomness in the system to facilitate synchronization for variance reduction (Law and Kelton 2000).

In this note, we propose a package for uniform random-number generation with multiple streams of (pseudo)random numbers and convenient tools to move around within and across these streams. The structure and the tools offered are similar to those in the package

Subject classifications: Simulation: random number generation, random variable generation. Statistical analysis. Computers/computer science: software.
Area of review: SIMULATION.

proposed by L'Ecuyer and Côté (1991) and L'Ecuyer and Andres (1997). The main differences are:

- The underlying "backbone" generator is more robust and has longer period than those used by these authors. We use the *combined multiple-recursive generator* (CMRG) `MRG32k3a` proposed by L'Ecuyer (1999a).
- The package proposed here has an object-oriented design. The *streams*, which can be seen as virtual RNGs, are declared at will, as instances of a *class*, instead of being numbered from 0 to N where N is fixed.

Other random number packages with multiple streams have been proposed in recent years; see, for example, Mascagni and Srinivasan (2000). These packages do not offer the same tools for streams and substreams as ours, and are not supported by the same theoretical analysis for the quality and independence of the different streams.

In what follows, we provide some background on the backbone CMRG, explain how the streams and substreams are defined, and give a C++ interface to the package. Implementations in C, C++, and Java, as well as a more detailed version of this paper, are available in the online companion to this paper on the *Operations Research* Web site ⟨http://or.pubs.informs.org/pages/collect.html⟩, and at ⟨http://www.iro.umontreal.ca/~lecuyer⟩.

1. DESCRIPTION AND IMPLEMENTATION OF THE SOFTWARE

1.1. The Underlying Backbone Generator

L'Ecuyer (1999a) gave several good parameter sets for CMRGs of different sizes. We have selected one of them, called `MRG32k3a`, as our backbone generator. It has two components each of order 3. At step n, its state is the pair of vectors $s_{1,n} = (x_{1,n}, x_{1,n+1}, x_{1,n+2})$ and $s_{2,n} = (x_{2,n}, x_{2,n+1}, x_{2,n+2})$, which evolve according to the linear recurrences

$$x_{1,n} = (1403580 \times x_{1,n-2} - 810728 \times x_{1,n-3}) \bmod m_1,$$

$$x_{2,n} = (527612 \times x_{2,n-1} - 1370589 \times x_{2,n-3}) \bmod m_2,$$

where $m_1 = 2^{32} - 209 = 4294967087$ and $m_2 = 2^{32} - 22853 = 4294944443$, and its output u_n is defined by

$$z_n = (x_{1,n} - x_{2,n}) \bmod 4{,}294{,}967{,}087,$$

$$u_n = \begin{cases} z_n/4294967088 & \text{if } z_n > 0, \\ 4294967087/4294967088 & \text{if } z_n = 0. \end{cases}$$

Its period length is $\rho = (m_1^3 - 1)(m_2^3 - 1)/2 \approx 2^{191} \approx 3.1 \times 10^{57}$. RNGs with much longer periods are also available, but their states must contain more bits and are therefore more expensive to manipulate. We think that our choice is a reasonable compromise. The parameters have been chosen so that the period is long, a fast implementation is available (in floating point arithmetic), and the generator performs well with respect to the spectral test in up to (at least) 45 dimensions. The spectral test in t dimensions measures the uniformity of the point set

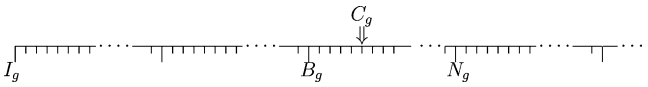
$$T_t = \{(u_0, \ldots, u_{t-1}) | (x_{1,0}, x_{1,1}, x_{1,2}) \in \mathbf{Z}_{m_1}^3, \; (x_{2,0}, x_{2,1}, x_{2,2}) \in \mathbf{Z}_{m_2}^3\},$$

where $\mathbf{Z}_m = \{0, \ldots, m-1\}$, and makes sure that this set covers the t-dimensional unit hypercube very uniformly. This T_t is the set of all overlapping t-tuples of successive values produced by the generator, from all possible initial states.

1.2. Multiple Streams and Substreams

Let ρ be the period length of the RNG and T its transition function; that is, $T(s_n) = s_{n+1}$, where s_n is the generator's state at step n, and $T^\rho(s) = s$. To partition the generator's sequence into disjoint streams and substreams, we choose two positive integers v and w, and let $z = v + w$. We first cut the long cycle into adjacent *streams* of length $Z = 2^z$ and then partition each of these streams into $V = 2^v$ *blocks* (or *substreams*) of length $W = 2^w$.

If s_0 is the initial seed of the generator and I_g denotes the *initial state* of stream g for $g \geqslant 1$, then we have $I_1 = s_0$, $I_2 = T^Z(s_0), \ldots, I_g = T^Z(I_{g-1}) = T^{(g-1)Z}(s_0), \ldots$. The first *substream* of stream g starts in state I_g, the second in state $T^W(I_g)$, the third in state $T^{2W}(I_g)$, and so on. At any moment during program execution, 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., of the *current substream*, and $N_g = T^W(B_g)$ the starting state of the *next substream*. In Figure 1, for example, the state of stream g is at the 6th value of the third substream, i.e., $2W + 5$ steps ahead of its initial state I_g and 5 steps ahead of B_g.



Whenever a new stream is created (instantiated), say the gth stream, the software automatically computes $I_g = T^Z(I_{g-1})$ and puts $C_g = B_g = I_g$. When going from a substream to the next one, the software must compute $N_g = T^W(B_g)$. Of course, W and Z must be large numbers, so a quick way to compute $s_{n+\nu}$ from s_n for large integers ν, without generating the intermediate values, must be available. For a combined MRG, we can do this for each of its components separately, as explained in L'Ecuyer (1990): One can write $s_{j,n+1} = A_j s_{j,n} \bmod m_j$ for some 3×3 matrix A_j, and then $s_{j,n+\nu} = (A_j^\nu \bmod m_j) s_{j,n} \bmod m_j$. The matrix $A_j^\nu \bmod m_j$ is computed via a standard divide-and-conquer algorithm (Knuth 1998), and can be precomputed once for $\nu = W$ and $\nu = Z$.

1.3. Choice of v and w

We have selected $v = 51$ and $w = 76$, so $W = 2^{76}$ and $Z = 2^{127}$. To select v and w, a spectral test for the

vectors of *nonsuccessive* output values spaced $h = 2^l$ steps apart was performed for different integer values of l, and we chose v and w so that the behavior was good for $l = v$, $l = w$, and $l = v + w$. More specifically, let $T_t(s, h)$ be the point set obtained if we replace $(u_n, \ldots, u_{n+t-1})$ by the first t components of the sequence $(u_n, \ldots, u_{n+s-1}, u_h, \ldots, u_{n+h+s-1}, u_{n+2h}, \ldots, u_{n+2h+s-1}, \ldots)$ in the definition of T_t. If the streams are started h apart, the points of $T_t(s, h)$ are those obtained by taking s successive values from the first stream, s successive values from the second stream, and so on until t values have been taken. We looked for values of l such that for $h = 2^l$, the point set $T_t(s, h)$ was very uniformly distributed, according to the spectral test, for all $s \leqslant 16$ and $t \leqslant 32$. This was done for $51 \leqslant l \leqslant 150$, and we found that the uniformity was particularly good for $l = 51, 76$, and 127.

2. A C++ INTERFACE TO THE PACKAGE RNGSTREAMS

We now describe the main public members of the class `RngStream` in C++. Other methods and further details are available at the INFORMS home page in the *Operations Research Online Collection* at ⟨http://or.pubs.informs.org/Pages/collect.html⟩.

```
class RngStream
{
public:
```

`RngStream (const char *name = "");`

This constructor creates a new stream with (optional) descriptor `name`. It initializes its seed I_g, and sets B_g and C_g to I_g. The seed I_g is equal to the initial seed of the package if this is the first stream created; otherwise it is Z steps ahead of the seed of the most recently created stream.

`static void SetPackageSeed (const unsigned long seed[6]);`

Sets the initial seed s_0 of the package to the six integers in the vector `seed`. The first three integers in the seed must all be less than $m_1 = 4294967087$, and not all 0; and the last three integers must all be less than $m_2 = 4294944443$, and not all 0. If this method is not called, the default initial seed is (12345, 12345, 12345, 12345, 12345, 12345).

`void ResetStartStream ();`

Reinitializes the stream to its initial state: C_g and B_g are set to I_g.

`void ResetStartSubstream ();`

Reinitializes the stream to the beginning of its current substream: C_g is set to B_g.

`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.

`void SetAntithetic (bool a);`

If `a = true`, the stream will start generating antithetic variates, i.e., $1 - U$ instead of U, until this method is called again with `a = false`.

`void IncreasedPrecis (bool incp);`

After calling this method with `incp = true`, each call to the generator (direct or indirect) for this stream will return a uniform random number with more bits of resolution (53 bits if machine follows IEEE 754 standard) instead of 32 bits, and will advance the state of the stream by two steps instead of one. More precisely, if `s` is a stream of the class `RngStream`, in the nonantithetic case, the instruction "`u = s.RandU01()`" will be equivalent to "`u = (s.RandU01() + s.RandU01() * fact)%1.0`" where the constant `fact` is equal to 2^{-24}. This also applies when calling `RandU01` indirectly (e.g., via `RandInt`, etc.). By default, or if this method is called again with `incp = false`, each call to `RandU01` for this stream advances the state by one step and returns a number with 32 bits of resolution.

`void WriteState () const;`

Writes (to standard output) the current state C_g of this stream.

`double RandU01 ();`

Normally, returns a (pseudo)random number from the uniform distribution over the interval $(0, 1)$, after advancing the state by one step. The returned number has 32 bits of precision in the sense that it is always a multiple of $1/(2^{32} - 208)$. However, if `IncreasedPrecis(true)` has been called for this stream, the state is advanced by two steps and the returned number has 53 bits of precision.

`long RandInt (long i, long j);`

Returns a (pseudo)random number from the discrete uniform distribution over the integers $\{i, i + 1, \ldots, j\}$. Makes one call to `RandU01`.

```
};
```

ACKNOWLEDGMENT

This work was supported by NSERC-Canada grant number ODGP0110050 to the first author.

REFERENCES

Knuth, D. E. 1998. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed. Addison-Wesley, Reading, MA.

Law, A. M., W. D. Kelton. 2000. *Simulation Modeling and Analysis*, 3rd ed. McGraw-Hill, New York.

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

——. 1999a. Good parameters and implementations for combined multiple recursive random number generators. *Oper. Res.* **47**(1) 159–164.

——. 1999b. Tables of maximally equidistributed combined LFSR generators. *Math. Comput.* **68**(225) 261–269.

——, T. H. Andres. 1997. A random number generator based on the combination of four LCGs. *Math. Comput. Simulation* **44** 99–107.

——, S. Côté. 1991. Implementing a random number package with splitting facilities. *ACM Trans. Math. Software* **17**(1) 98–111.

——, R. Simard. 2001. On the performance of birthday spacings tests for certain families of random number generators. *Math. Comput. Simulation* **55**(1–3) 131–137.

Mascagni, M., A. Srinivasan. 2000. Algorithm 806: SPRNG: A scalable library for pseudorandom number generation. *ACM Trans. Math. Software* **26** 436–461.

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