ABSTRACT

We study a class of combined multiple recursive random number generators constructed in a way that each component runs fast and is easy to implement, while the combination enjoys excellent structural properties as measured by the spectral test. Each component is a linear recurrence of order $k > 1$, modulo a large prime number, and the coefficients are either 0 or are of the form $a_1 \cdot 2^q \cdot 2^r$. This allows a simple and very fast implementation, because each modular multiplication by a power of 2 can be implemented via a shift, plus a few additional operations for the modular reduction. We select the parameters in terms of the performance of the combined generator in the spectral test. We provide a specific implementation.

1 INTRODUCTION: MULTIPLE RECURSIVE GENERATORS

1.1 Multiple Recursive Generators

A multiple recursive generator (MRG) is defined by the linear recurrence

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \mod m;$$

$$u_n = x_n / m. \quad (1)$$

The modulus $m$ and the order $k$ are positive integers, the coefficients $a_i$ belong to $\mathbb{Z}_m = \{0, 1, \ldots, m - 1\}$, and the state at step $n$ is the vector $x_n = (x_{n-k+1}, \ldots, x_n)$. The maximal period length of the recurrence (1) is $\rho = m^k - 1$, and this length is attained if and only if $m$ is prime and the characteristic polynomial of the recurrence,

$$P(z) = z^k - a_1 z^{k-1} - \cdots - a_k,$$

is a primitive polynomial modulo $m$. Primitive polynomials can be found by random search as explained in L’Ecuyer, Blouin, and Couture (1993) and L’Ecuyer (1999a). To obtain a primitive polynomial, one needs at least 2 non-zero coefficients $a_i$’s. This minimal number of non-zero coefficients yields the following economical version of (1):

$$x_n = (a_r x_{n-r} + a_k x_{n-k}) \mod m. \quad (3)$$

The classical linear congruential generator (LCG) is obtained with $k = 1$. Further details about MRGs can be found, e.g., in Knuth (1998), Niederreiter (1992), L’Ecuyer (1994), L’Ecuyer (1996), and the references therein.

Besides period length, two important issues in the design of MRGs are the statistical quality and the efficiency of the implementation.

The statistical quality is traditionally assessed by measuring the uniformity of the set of all vectors of successive output values $(u_n, \ldots, u_{n+t-1})$, from all initial states, as we explain in Section 2. For the implementation, the key issue is how to compute efficiently the products $a_i \cdot x \mod m$ when $m$ is large.

1.2 Implementation

A first approach for computing $a x \mod m$ is approximate factoring (Bratley, Fox, and Schrage 1987; L’Ecuyer and Côté 1991). It uses integer arithmetic and a clever decomposition of $m$. It works if $a^2 < m$ or if $a = [m/i]$ where $i^2 < m$, and if all integers between $-m$ and $m$ are well represented on the computer.

A second approach computes the product and the division by $m$ (for the mod operation) directly in floating-point arithmetic. On computers that obey the IEEE 64-bit floating-point standard (most computers do), all integers up to $2^{53}$ are represented exactly in floating point, and the floating-point implementation works if $am < 2^{53}$. See L’Ecuyer (1999a) for details and examples.

A third approach, introduced by Wu (1997) and generalized by L’Ecuyer and Simard (1999), which we call the powers-of-2 decomposition, assumes that $a$ is a sum or a difference of a small number of powers of 2, e.g.,...
A direct efficient implementation of the recurrence (1) can be implemented by a left shift of the binary representation of $x$, and the product $ax$ is computed by adding and/or subtracting. The details are given in Section 3. This approach turns out to be more efficient than the other two, according to our experiments.

Note that replacing any $a_i$ by $a_i \pm m$ changes nothing to the recurrence (1). If for some $a_i \in \mathbb{Z}_m$, $|a_i - m|$ satisfies one of the above conditions whereas $a_i$ does not satisfy that condition, then one can replace $a_i$ by $\tilde{a}_i = a_i - m$ when implementing (1). This is equivalent to allowing negatives values for the $a_i$'s, which we shall do in the remainder of this paper.

### 1.3 Combined MRGs

A direct efficient implementation of the recurrence (1) can generally be obtained only when the number of non-zero coefficients $a_i$ is small, and when special conditions are imposed on these coefficients, as explained in the previous subsection. However, imposing these constraints usually implies that the resulting MRG has a poor lattice structure (L’Ecuyer 1997; L’Ecuyer 1999a). In particular, good behavior is possible only if the sum of squares of the $a_i$ is large (L’Ecuyer 1997).

This has motivated the introduction of combined MRGs, which are constructed so that the components are easy to implement efficiently while the structure of the resulting combined generator has good quality. L’Ecuyer (1996) has proposed and analyzed the following class of combined MRGs, based on $J$ linear recurrences, with the same order $k$ and distinct prime moduli $m_j$, running in parallel. For $1 \leq j \leq J$, let

$$x_{j,n} = (a_{j,1}x_{j,n-1} + \cdots + a_{j,k}x_{j,n-k}) \mod m_j \quad (3)$$

and suppose that the recurrence (1.3) has period $\rho_j = m_j^k - 1$. Suppose also that the least common multiple of $\rho_1, \ldots, \rho_J$ is $\rho = \rho_1 \cdots \rho_J / 2^{J-1}$. (This is the best that one can do, because each $\rho_j$ is necessarily even.) Define the two combinations

$$z_n = \left( \sum_{j=1}^{J} \delta_j x_{j,n} \right) \mod m_1; \quad u_n = z_n / m_1 \quad (3)$$

and

$$w_n = \left( \sum_{j=1}^{J} \frac{\delta_j x_{j,n}}{m_j} \mod 1 \right) \quad (3)$$

where the $\delta_j$'s are integers such that $\delta_j$ is relatively prime with $m_j$ for each $j$. L’Ecuyer (1996) has shown that the sequence $\{w_n, n \geq 0\}$ defined by (1.3) is the same as the sequence $\{u_n, n \geq 0\}$ produced by the MRG (1–2), with $m = m_1 \cdots m_J$, and explains how to compute the corresponding $a_i$'s. Moreover, the numbers $u_n$ and $w_n$ produced by (1.3) and (1.3), respectively, differ only by a very small quantity when the $m_j$ are close to each other. Explicit bounds on the difference are given in L’Ecuyer (1996).

The generators considered in this paper are combined MRGs of this form, constructed so that the structure of the combined MRG has excellent quality while (1.3) can be implemented efficiently for each $j$. This was already achieved by L’Ecuyer (1996) and L’Ecuyer (1999a) for implementations based on approximate factoring and on floating-point arithmetic, respectively. The aim of this paper is to propose combined MRGs that are faster for an equivalent statistical quality, by using the powers-of-2 decomposition method.

### 1.4 Overview of the Remainder

The remainder of the paper is organized as follows. In Section 2, we recall the quality criteria for selecting MRGs based on an analysis of their lattice structure by the spectral test. In Section 3, we describe the implementation method for coefficients that are a sum or a difference of a small number of powers of 2. In Section 4, we explain how we searched for good single and combined MRGs with coefficients of this form, and why we prefer the combined MRGs over the non-combined ones. In Section 5, we give a specific implementation of a combined MRG of this form and we compare its speed with other combined MRGs proposed in L’Ecuyer (1996) and L’Ecuyer (1999a), and implemented via approximate factoring and floating point arithmetic.

### 2 LATTICE STRUCTURE AND QUALITY CRITERIA

Let

$$\Psi_t = \{ (u_0, \ldots, u_{t-1}) : (x_0, \ldots, x_{k-1}) \in \mathbb{Z}_m^k \},$$

the set of all vectors of $t$ successive output values of the MRG (1–2), from all possible initial states. If the initial seed of the MRG is chosen at random, this $\Psi_t$ is viewed in a sense as the sample space from which points are chosen at random to approximate the uniform distribution over the $t$-dimensional unit hypercube $[0, 1)^t$. This means that the generator should be constructed so that $\Psi_t$ covers $[0, 1)^t$ very evenly, for $t$ up to some arbitrary number.

It is well known that the set $\Psi_t$ for an MRG is equal to the intersection of a lattice $L_t$ with the unit hypercube $[0, 1)^t$ (Knuth 1998; L’Ecuyer and Couture 1997). This implies that $\Psi_t$ lies on a limited number of equidistant parallel hyperplanes, at a distance $d_t$ apart, where $1/d_t$ turns out to be equal to the Euclidean length of the shortest
We want to compute possible. Computing \( dt \) and \( x \) by \( x \) nonzero vector in the dual lattice of \( L_t \), defined as the set of vectors in \( \mathbb{R}^t \) whose scalar product by any vector of \( L_t \) is an integer. Computing \( dt \) is called the spectral test (Knuth 1998; L’Ecuyer and Couture 1997). For \( \Psi_t \) to be evenly distributed over \([0, 1]^t\), we want \( dt \) to be small.

Here, we use the same figure of merit as in L’Ecuyer (1999a), namely

\[
M_t = \min_{t \leq t_1} d_t^* (m^k) / d_t,
\]

where \( t_1 > k \) is a selected constant (the maximal dimension that is considered) and \( d_t^* (m^k) = 1 / \rho_t m^{k/t} \) is an absolute lower bound on \( d_t \), for given \( k \) and \( t \). For \( t \leq 8 \), we take \( \rho_t \) as the value of \( \gamma_t \) defined in Knuth (1998), page 109, whereas for \( t > 8 \), we take \( \rho_t = \exp[R(t)/t] \) where \( R(t) \) is the bound of Rogers on the density of sphere packings (Conway and Sloane 1999; L’Ecuyer 1999b). This \( M_t \) is always between 0 and 1 and we want it to be as large as possible.

We recall that a general upper bound on \( 1/d_t^2 \) is given by

\[
1/d_t^2 \leq 1 + \sum_{i=1}^{k} a_i^2,
\]

which means that a necessary condition for good quality is that the sum of squares of the coefficients must be large.

3 IMPLEMENTATION BY THE POWERS-OF-2 DECOMPOSITION METHOD

We want to compute

\[
y = 2^q x \mod m
\]

where \( 0 < x < m \). We decompose \( m \) and \( x \) as \( m = 2^e - h \) and \( x = x_0 + 2^{e-q} x_1 \), where \( h > 0 \), \( x_0 = x \mod 2^{e-q} \), and \( x_1 = [x/2^{e-q}] \). We then have

\[
y = (2^q x_0 + 2^{e-q} x_1) \mod (2^e - h) = (2^q x_0 + h x_1) \mod (2^e - h). \tag{3}
\]

We assume that the following inequalities hold:

\[
h < 2^q \quad \text{and} \quad h(2^q - (h + 1) 2^{-e+q}) < m. \tag{3}
\]

Under these conditions, each of the two terms \( 2^q x_0 \) and \( h x_1 \) in (3) is less that \( m \), and \( y \) can be computed as follows (L’Ecuyer and Simard 1999): Shift the binary representation of \( x_0 \) by \( q \) positions to the left to obtain \( 2^q x_0 \), add \( h \) times \( x_1 \), and subtract \( m \) if the result exceeds \( m - 1 \). This can be implemented using unsigned integers and the intermediate results will never exceed \( 2m - 1 \). The procedure requires a single multiplication, between \( h \) and \( x_1 \).

To multiply \( x \) by \( a = \pm 2^q \pm 2^r \) modulo \( m \), repeat the procedure with \( r \) instead of \( q \), and add (or subtract) the results modulo \( m \). L’Ecuyer and Simard (1999) give an implementation in C.

Note that if \( q = 0 \) in (3), nothing needs to be done \((y = x)\), whereas if \( q = 1 \), one can simply add \( x + x \), and subtract \( m \) if the result exceeds \( m - 1 \). We will exploit these special cases when we will select the parameters of our combined MRGs.

Wu (1997) introduced this method for the special case where \( h = 1 \). In this case, one obtains \( y = 2^q x_0 + x_1 \), which means that the binary representation of \( y \) is obtained simply by exchanging the blocks of bits \( x_0 \) and \( x_1 \) in the binary representation of \( x \), i.e., rotating the bits by \( q \) positions. This simple rotation does not change the bits of \( x \) very much. L’Ecuyer and Simard (1999) have shown that as a result, the Hamming weights of \( x \) and of \( ax \mod m \) (i.e., the number of 1’s in their respective binary representations) tend to be strongly dependent when \( m \) and \( a \) have the form \( m = 2^e - 1 \) and \( a = \pm 2^q \pm 2^r \). For \( k = 1 \) (i.e., LCGs), they showed that this dependence also appears between the number of 1’s in the binary representations of two successive output values \( u_{n-1} \) and of \( u_n \), and they proposed a simple statistical test to detect it. The specific LCGs proposed by Wu (1997) fail this independence test decisively, whereas LCGs whose multipliers have a more complicated binary representation typically pass this test.

LCGs with multipliers of the form \( a = \pm 2^q \pm 2^r \) have in fact been proposed and used a long time ago: The infamous RANDU generator (IBM 1968) has indeed \( a = 2^{16} - 2 + 1 \) and \( m = 2^{32} \). These parameters were selected for the ease of implementation, but led to important deficiencies in the generator’s structure (see, e.g., Law and Kelton 2000).

4 SEARCH FOR GOOD PARAMETERS

We performed computer searches to find good single and combined MRGs that can be implemented via the powers-of-2 decomposition method.

The first search was for MRGs of order \( k = 6 \), with modulus \( m = 2^{31} - 1 \). With this \( m \), we have \( h = 1 \) and we thus avoid the multiplication by \( h \): We are back to the special case considered by Wu (1997). We imposed the condition that each coefficient \( a_i \) had to be of the form

\[
a = \pm 2^q \pm 1, \quad \text{or} \quad a = \pm 2^q, \quad \text{or} \quad a = 0. \tag{3}
\]

Even with these conditions, an exhaustive search would be too long, so we made a random search. Almost all of the generators that we examined and that satisfied these conditions had a very bad lattice structure in dimension 7, 8 or 9. These generators typically had a good behavior in higher dimensions. The best generator that we found, based on the criterion \( M_{16} \), has \( M_{16} = 0.25012 \). This is not very
good. Its coefficients are \( a_1 = 2^{15}, a_2 = 0, a_3 = -2^9 + 1, a_4 = 2^{20} - 1, a_5 = -2^5 - 1, \) and \( a_6 = 2^{26} - 1. \) We call it MRG31k6s. It has 5 non-trivial powers of two in its coefficients, so it can be compared to MRG31k3p, to be presented in the next section, in terms of the number of multiplications by powers of two. We cannot recommend it, however, because its lattice structure is relatively poor and, perhaps more importantly, because the small number of powers of 2 in the coefficients means (intuitively) that there is not much mixing of the bits, similar to, e.g., the generator of Wu 1997 (although not as bad).

In our second search, with the same \( k \) and \( m \), we allowed coefficients with more non-trivial powers of 2. The condition on the coefficients was that they had to be of the form \( a = \pm 2^q \pm 2^r \). MRGs with good lattice structures were much easier to find under these relaxed conditions. The best generator that we found, based on \( M_{16} \), has \( M_{16} = M_{18} = 0.59149. \) We call it MRG31k6l. Its coefficients are \( a_1 = 2^{23} + 2^{16}, a_2 = 2^{19} - 2^{12}, a_3 = 2^{27} + 2^{15}, a_4 = -2^{10} - 2^7, a_5 = -2^9 - 1, \) and \( a_6 = 2^{27} + 2^{16}. \)

Of course, this generator will be slower than MRG31k3p, because there is more multiplications by powers of 2 to perform. We will compare their speeds at the end of the next section.

Our third search was for combined MRGs with 2 components of order \( k = 3 \), with moduli \( m_1 = 2^{31} - 1 \) and \( m_2 = 2^{31} - 21069 \), and with some coefficients equal to zero and the others of the form \( \pm 2^q \) or \( \pm 2^q \pm 1. \) We performed a random search. For each coefficient of each component, we specified the desired form: either 0, or \( \pm 2^q \), or \( \pm 2^q \pm 1. \) Each coefficient received randomly between 1 to 10 possible values, depending on the specified form. We retained only the coefficients for which the inequalities (3) were satisfied, and only the recurrences (or characteristic polynomials) that satisfied the maximal period conditions. We then examined all the possible combinations of one MRG component of each type, among those retained, to find out which combined generator performed best on spectral test. These combined MRG have approximatively the same period length as the MRGs of order 6 in our first two searches. The best combined MRG that we found, MRG31k3p, is described in the next section.

5  A SPECIFIC GENERATOR AND SOME TIMINGS

In our third search, we found the following combined MRG with \( J = 2 \) components of order \( k = 3 \), whose lattice structure is good at least up to 48 dimensions, with \( M_{18} = 0.60159. \) The two components are defined by the parameters

\[
\begin{align*}
m_1 &= 2^{31} - 1 = 2147483647 \\
a_{11} &= 0
\end{align*}
\]

Thus, each component has only 2 nonzero coefficients, one of them of the form \( a_{ij} = 2^q \) and the other one of the form \( a_{ij} = 2^q + 1. \) This will simplify the implementation.

The combination (1.3) is exactly equivalent to an MRG of order 3 with parameters

\[
\begin{align*}
m &= 4611640770946945613 \\
a_1 &= 4341088847531259234 \\
a_2 &= 2349160800583431525 \\
a_3 &= 392781859046737243.
\end{align*}
\]

This generator has 2 distinct cycles of length \( \rho = m_1m_2/2 \approx 2^{185}. \)

Figure 1 gives an implementation of this generator in the C language. In this code, the bit masks \( \text{mask}^* \) are used to separate the bits of the \( x_j;i \)’s in 2 blocks as explained in Section 3. For example, \( \text{mask12} \) contains 23 zeros followed by 9 ones. So, in the first line of the code for the first component, the statement

\[
((\text{x11} \& \text{mask12}) << 22) + (\text{x11} >> 9)
\]

extracts the 9 least significant bits of \( \text{x11} \), shifts them to the left by 22 positions, and adds this to \( \text{x11} \) shifted to the right by 9 positions. The result is the product of \( \text{x11} \) by \( a_{12} = 2^{22} \), modulo \( m_1 = 2^{31} - 1. \) The other products are implemented in a similar way. The several instructions of the form

\[
\text{if} (y2 > m2) y2 -= m2;
\]

are necessary to avoid overflow. Indeed, since the terms added are between 0 and 2\(^{31} - 1 \) and since unsigned integers cannot exceed 2\(^{32} - 1 \), we cannot safely add more than 2 terms at a time without reducing the sum modulo \( m_j \).

To have an idea of the speed improvement of this new generator over the previous combined MRG implementations, for each generator we generated 10 million (10\(^7 \)) random numbers and added them up, looked at how much CPU time it took (user time + system time), and printed the sum (this may be convenient for checking correctness of an implementation). In all cases, each integer in the seed was 12345. We did this on a 64-bit SUN Ultra-2/160 using the system’s compiler (cc, version 4.2) with the “-fast -xtarget=ultra -xarch=v8plusa” options and with the GNU C (gcc)


```c
#define m1 2147483647
#define m2 2147462579
#define norm 4.656612873077393e-10
#define mask12 511
#define mask13 16777215
#define mask21 65535

unsigned long x10, x11, x12, x20, x21, x22;

double MRG31k3p ()
{
    register unsigned long y1, y2; /* For intermediate results */

    /* First component */
    y1 = (((x11 & mask12) << 22) + (x11 >> 9))
        + (((x12 & mask13) << 7) + (x12 >> 24));
    if (y1 > m1) y1 -= m1;
    y1 += x12;
    if (y1 > m1) y1 -= m1;
    x12 = x11; x11 = x10; x10 = y1;

    /* Second component */
    y1 = ((x20 & mask21) << 15) + 21069 * (x20 >> 16);
    if (y1 > m2) y1 -= m2;
    y2 = ((x22 & mask21) << 15) + 21069 * (x22 >> 16);
    if (y2 > m2) y2 -= m2;
    y2 += x22;
    if (y2 > m2) y2 -= m2;
    y2 += y1;
    if (y2 > m2) y2 -= m2;
    x22 = x21; x21 = x20; x20 = y2;

    /* Combinaison */
    if (x10 <= x20) return ((x10 - x20 + m1) * norm);
    else return ((x10 - x20) * norm);
}
```

Figure 1: Implementation of a combined MRG with the Powers-of-2 Decomposition Method

and GNU C++ (g++) compilers, with the options "-O3 -ffast-math -fexpensive-optimizations -finline-functions". Then we did the same on two other computers, with a 500 MHz Pentium III processor and a 750 MHz AMD Athlon processor, respectively, both under the Red Hat Linux operating system, with the GNU C and GNU C++ compilers, with the same options as on the SUN.

The timings for the selected generators, in seconds, are given in Table 1, for the C compilers. The timings for the C++ compilers are practically identical to those with the corresponding C compiler. In the table, we also indicate the period length, the type of implementation (AF for approximate factoring, FP for floating-point, and P2D for powers-of-2 decomposition), and the sum of the $10^7$ numbers generated. The generator MRG31k3p is that of Figure 1, MRG31k6s and MRG31k6l were introduced in the previous section, MRG32k3a is the combined MRG proposed in L’Ecuyer (1999a), combMRG96a is that given in Figure I of L’Ecuyer (1996), and combMRG96b is a variant of combMRG96a with the moduli and multipliers defined as constants in the code instead of variables as in combMRG96a.

Aside from MRG31k3s, which we discard because of its poor performance in the spectral test, the generator MRG31k3p in the first line of the table is the fastest on all processors, except on the SUN with the GNU compiler, where MRG32k3a wins by a small margin. This illustrates the fact that speed comparisons depend heavily on compilers and machine architecture.

The generator of Figure 1 gives only 31 bits of precision even though it returns 53-bit floating-point numbers. If more precision is desired, one can combine two successive numbers produced by the generator to construct each output number.
Table 1: CPU time (seconds) to generate and add $10^7$ random numbers, and value of the sum

<table>
<thead>
<tr>
<th>RNG</th>
<th>Period length $\approx$</th>
<th>Method</th>
<th>SUN Ultra-2 cc</th>
<th>Pentium-III 500 MHz gcc</th>
<th>AMD Athlon 750 MHz gcc</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRG31k3p</td>
<td>$2^{185}$</td>
<td>P2D</td>
<td>3.3</td>
<td>2.8</td>
<td>1.4</td>
<td>5000214.81</td>
</tr>
<tr>
<td>MRG31k6s</td>
<td>$2^{186}$</td>
<td>P2D</td>
<td>3.4</td>
<td>2.8</td>
<td>1.3</td>
<td>4999947.37</td>
</tr>
<tr>
<td>MRG31k6l</td>
<td>$2^{186}$</td>
<td>P2D</td>
<td>5.5</td>
<td>4.5</td>
<td>2.0</td>
<td>5000070.98</td>
</tr>
<tr>
<td>MRG32k3a</td>
<td>$2^{191}$</td>
<td>FP</td>
<td>5.1</td>
<td>5.6</td>
<td>2.3</td>
<td>5001090.95</td>
</tr>
<tr>
<td>combMRG96a</td>
<td>$2^{185}$</td>
<td>AF</td>
<td>18.2</td>
<td>6.0</td>
<td>3.7</td>
<td>4999897.05</td>
</tr>
<tr>
<td>combMRG96b</td>
<td>$2^{185}$</td>
<td>AF</td>
<td>11.5</td>
<td>6.0</td>
<td>3.3</td>
<td>4999897.05</td>
</tr>
</tbody>
</table>

value. For example, if MRG31k3p outputs the sequence $u_1, u_2, \ldots$ one can use the sequence $v_1, v_2, \ldots$ of pseudorandom numbers defined by $v_i = (u_{2i} + u_{2i-1}) \mod 1$ for some constant $v$ between $2^{-21}$ and $2^{-32}$.

6 CONCLUSION

Combined MRGs with multipliers of the form $\pm 2^q \pm 2^r$ are the fastest good-quality MRGs available to date, when comparing generators having approximately the same period length. These combined MRG possess good theoretical properties in terms of their period length and the quality of their lattice structure, and behave well in empirical statistical tests. In the future, we plan to search for good generators of this form by applying the spectral test not only to the vectors of successive output values produced by the generator (as usual), but to certain vectors of non-successive output values as well, as suggested by L’Ecuyer and Lemieux (2000) in the context of selecting lattice rules for quasi-Monte Carlo integration. These combined MRGs could also be combined with small, efficient, nonlinear generators to destroy the (linear) lattice structure and we intend to analyze such combinations.

ACKNOWLEDGMENTS

This work has been supported by NSERC-Canada grant No. ODGP0110050 and FCAR-Québec Grant No. 00ER3218 to the first author, and via an NSERC-Canada scholarship to the second author.

REFERENCES

Wu, P.-C. 1997. Multiplicative, congruential random number generators with multiplier $\pm 2^{61} \pm 2^{62}$ and modulus

688

AUTHOR BIOGRAPHIES

PIERRE L’ECUYER is a professor in the “Département d’Informatique et de Recherche Opérationnelle”, at the University of Montreal. He received a Ph.D. in operations research in 1983, from the University of Montréal. He obtained the E. W. R. Steacie grant from the Natural Sciences and Engineering Research Council of Canada for the period 1995–97. His main research interests are random number generation, quasi-Monte Carlo methods, efficiency improvement via variance reduction, sensitivity analysis and optimization of discrete-event stochastic systems, and discrete-event simulation in general. He is an Area Editor for the ACM Transactions on Modeling and Computer Simulation. More details at: <http://www.iro.umontreal.ca/~lecuyer>, where his recent research articles are available on-line.

RENÉE TOUZIN is currently an M.Sc. student in the “Département d’Informatique et de Recherche Opérationnelle”, at the University of Montréal. She works on the design and analysis of multiple recursive random number generators.