Version: November 25, 1997

A Random Number Generator
Based on the Combination of Four LCGs

Pierre L'ECUYER
Université de Montréal

Terry H. ANDRES
Atomic Energy of Canada Limited

ABSTRACT: A portable package for uniform random number generation is proposed, based on a backbone generator with period length near 2^{121}, which is a combination of four linear congruential generators. The package provides for multiple (virtual) generators evolving in parallel. Each generator also has many disjoint subsequences, and software tools are provided to reset the state of any generator to the beginning of its first, previous, or current subsequence. Such facilities are helpful to maintain synchronization for implementing variance reduction methods in simulation. Computer implementations are available in the C and Modula-2 languages.

Keywords: Random number generation, jump-ahead, software package.

Authors' Addresses:

P. L'Ecuyer, Département d'Informatique et de Recherche Opérationnelle (IRO), Université de Montréal, C.P. 6128, Succ. Centre-Ville, Montréal, H3C 3J7, Canada;

e-mail: `lecuyer@iro.umontreal.ca`

www: `http://www.iro.umontreal.ca/~lecuyer`

T. Andres, AECL Whiteshell Laboratories

Pinawa, Manitoba, R0E 1L0, Canada;

e-mail: `andrest@wl.aecl.ca`

1 Introduction

Stochastic simulation programming languages and environments usually provide for several uniform random number *sequences*, often numbered 1, 2, 3, ... Almost always, all these sequences are from the same underlying random number generator, but using different starting points (or *seeds*) in the main sequence. L'Ecuyer and Côté [9] describe a package with 32 (virtual) generators, with seeds spaced 2^{50} values apart. It is based on a combined linear congruential generator (LCG) with period length $\rho \approx 2^{61}$, previously proposed by L'Ecuyer [4]. Each generator corresponds to a subsequence of 2^{50} values, which is further split into $V = 2^{20}$ segments of length $W = 2^{30}$. By a simple procedure call, any generator can jump ahead to either the beginning of its next segment, or its current segment, or its first (initial) segment. A computer implementation in Pascal is provided in their paper. Unfortunately, only 32 generators and these values of V and W are now deemed too small for many current and future applications.

In this paper, we provide a similar package, but based on a more robust backbone generator, with longer period ($\rho \approx 2^{121}$) and improved structural properties. This package also offers more generators, as well as more and longer segments. The values of V and W are powers of two which can be chosen by the user. The initial seed of the first generator can be set to any admissible value (a vector of four positive integers) and the initial seeds of the other generators are automatically recalculated by the package so that they remain VW values apart. This is implemented with efficient jump-ahead tools.

What is the use of such a package? Suppose one wants to compare two similar systems using simulation with common random numbers, with n simulation runs for each system. (For more details about common random numbers and other variance reduction techniques, see for example [1, 3].) To make sure that each pair of runs comparing the two systems uses the same random numbers for the same purpose, one would assign different generators to different parts of the system (e.g., for comparing two similar queueing systems, one stream for the interarrival times, one stream for the service times at each queue, one stream for routing decisions, etc.), and make sure that each generator produces the same sequence of numbers for the two systems. Without appropriate tools, this may be cumbersome, because the two systems do not necessarily use the same amount of random numbers. With the proposed package, one assigns each run number to the corresponding segment number, for all generators. With the first system, simulate run 1 with the initial seed, and before each new run, advance each generator to the beginning of the next segment. After the nth run, reset the generators to their initial seeds and do the same for the second system.

Another application is for parallel simulation on several processors. In this case, one uses a different (virtual) generator on each processor. These generators are the same except for their seeds. This makes the implementation much easier. For more background on random number generation see, for example, [2, 7, 8, 12].

In the next section, we explain how the backbone generator was chosen, based on a lattice structure analysis of its successive output values. We also perform a lattice structure analysis of the vectors of non-successive output values spaced 2^ℓ values apart, for different integer values of ℓ. Section 3 gives the functional definition of the package, and then the implementation, in the C language.

2 A Combination of Four LCGs

2.1 Combined LCGs

Wichmann and Hill [13] and L'Ecuyer [4] proposed slightly different methods for combining LCGs with distinct prime moduli. L'Ecuyer and Tezuka [11] later showed that a combined generator of the Wichmann and Hill type is in fact an LCG with modulus equal to the product of the moduli of the individual components, and that the combination method of [4] is approximately equivalent.

To be more specific, consider J LCGs, based on the recurrences

$$x_{j,n} = a_j x_{j,n-1} \bmod m_j,$$

for $j = 1, \ldots, J$. We assume that the moduli m_j are distinct primes and that the jth LCG has (maximal) period length $\rho_j = m_j - 1$. Let $\delta_1, \ldots, \delta_J$ be arbitrary integers such that for each j, δ_j and m_j have no common factor. Define the two combinations:

$$z_n = \left(\sum_{j=1}^{J} \delta_j x_{j,n} \right) \bmod m_1; \qquad u_n = z_n / m_1 \tag{1}$$

and

$$w_n = \left(\sum_{j=1}^{J} \frac{\delta_j x_{j,n}}{m_j} \right) \bmod 1. \tag{2}$$

Let $m = \prod_{j=1}^{J} m_j$. The following is proved in [11].

THEOREM **1.** *(i) The sequences $\{u_n\}$ and $\{w_n\}$ both have period length $\rho = lcm(\rho_1, \ldots, \rho_J)$ (the least common multiple of the ρ_j). Each ρ_j is even and the maximum possible value of ρ is $\rho = (m_1 - 1) \cdots (m_J - 1)/2^{J-1}$.*

(ii) The w_n obey the recurrence:

$$x_n = a x_{n-1} \bmod m; \qquad w_n = x_n/m, \tag{3}$$

where a can be computed by a formula given in [11] and does not depend on the δ_j.

(iii) One has $u_n = w_n + \epsilon_n$, with $\Delta^- \le \epsilon_n \le \Delta^+$, where Δ^- and Δ^+ can be computed as explained in [11] and are very small when the m_j are close to each other.

The combinations (1) and (2) are thus practical ways of implementing an LCG with large modulus m and multiplier a. The choice of parameters should then be based on a structural analysis of the corresponding LCG (3), rather than on the analysis of the individual components.

2.2 The Lattice Structure

For each integer $t \ge 0$, define

$$T_t = \{\boldsymbol{u}_n = (u_n, \ldots, u_{n+t-1}) \mid n \ge 0,\ x_0 \in \mathbb{Z}_m\}, \tag{4}$$

where $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$. This T_t is the set of all overlapping t-tuples of successive values produced by (3), with $u_n = x_n/m$, from all possible initial seeds. It turns out that T_t is equal to the intersection of a lattice L_t with the unit t-dimensional cube $[0, 1)^t$, which means that the points of T_t form a very regular structure (see, e.g., [2, 11, 10] for more details on this lattice structure). In particular, all these points lie on a relatively small number of equidistant parallel hyperplanes. For the points to be well spread within the unit cube, the distance d_t between the successive hyperplanes, for the family of hyperplanes covering all the points and which are farthest apart, must be small.

Computing d_t can be formulated as a quadratic integer optimization problem and is sometimes called the *spectral test* [2]. The value of d_t can be normalized to $S_t = d_t^*(m)/d_t$, where $d_t^*(m)$ is a lower bound on d_t for any t-dimensional lattice whose intersection with the unit hypercube $[0, 1)^t$ contains m points. This lower bound is known only for $t \leq 8$ (computing it for larger values is an open problem in the field of *Geometry of Numbers*). Another quality measure is the Beyer quotient q_t ([5, 10]), defined as the ratio of lengths of a shortest and longest vectors in a Minkowski-reduced basis for the lattice (the largest of such ratios if there is more than one Minkowski-reduced basis), and which should be close to one. Computing q_t is typically more costly than d_t. Both can be computed using the software package of [10].

2.3 Finding a Good Combination

Our goal was to construct a combined LCG with 4 components, with the following properties.

- Each component is an LCG with prime modulus m_j slightly smaller than 2^{31}, multiplier a_j such that $(m_j \bmod a_j) < m_j/a_j$ (for ease of implementation, as in [9]), and full period length $\rho_j = m_j - 1$.
- The moduli satisfy $m_j < m_{j-1} - 100$ for $j > 1$ and $(m_1 - 1)/2, \ldots, (m_4 - 1)/2$ have no common factor, so the combined generator has period length equal to the product of the ρ_j, divided by 8.
- The combined generator must have the best possible figure of merit

  $$M_8 = \min_{t \leq 8} S_t,$$

  measuring the quality of its lattice structure. Its values of d_t and q_t must also be reasonably good for t up to at least 25.

We made an intensive computer search to find a combined generator with these properties, and with the best possible value of M_8. We came up with the following parameters:

j	m_j	a_j
1	2147483647	45991
2	2147483543	207707
3	2147483423	138556
4	2147483323	49689

The LCG corresponding to the combined generator has modulus, multiplier, and period length

$$
\begin{aligned}
m &= 21267641435849934371830464348413044909, \\
a &= 5494569482908719143153333426731027229,
\end{aligned}
$$

and

$$\rho = (2^{31} - 2)(2^{31} - 106)(2^{31} - 226)(2^{31} - 326) \approx 2^{121}.$$

Table 1 gives the values of d_t and q_t up to dimensions 25, and the values of S_t up to dimension 8, for this combined generator. For comparison, Table 2 gives the corresponding values for the combined LCG with two components on which the package of [9] is based. In all dimensions t, the distances d_t between the hyperplanes are significantly smaller in Table 1 than in Table 2, so the new proposal is a clear improvement.

For the implementation given in the next section, we chose the combination (2) with $\delta_j = (-1)^{j+1}$. One can easily change the implementation to use different coefficients δ_j, or the other type of combination (1).

2.4 Lacunary Indices

The points formed by values with a fixed set of non-consecutive indices also belong to parallel hyperplanes which could be far apart. For instance, select two integers s and h, consider the set of infinite sequences:

$$
\begin{aligned}
T(s,h) &= \{\boldsymbol{u}_n = (u_n, \ldots, u_{n+s-1}, u_{n+h}, \ldots, u_{n+h+s-1}, \\
&\qquad u_{n+2h}, \ldots, u_{n+2h+s-1}, \ldots) \mid n \geq 0,\ x_0 \in \mathbb{Z}_m\}, \qquad (5)
\end{aligned}
$$

and let $T_t(s,h)$ be the set of t-dimensional points obtained by taking the first t coordinates of the points of $T(s,h)$. These points are those obtained by taking s successive values

Table 1: Lattice structure analysis for the proposed combination

t	d_t	q_t	S_t
2	2.37985E-19	0.82237	0.84792
3	4.18014E-13	0.65037	0.76924
4	5.23812E-10	0.75312	0.74755
5	3.41038E-08	0.78447	0.81535
6	6.31042E-07	0.79531	0.73773
7	4.72573E-06	0.78448	0.73112
8	2.09046E-05	0.59581	0.72992
9	7.78162E-05	0.66644	
10	2.29731E-04	0.56591	
11	4.65550E-04	0.67351	
12	7.88142E-04	0.64362	
13	1.61011E-03	0.57634	
14	2.02122E-03	0.74053	
15	2.93403E-03	0.86283	
16	4.82518E-03	0.69194	
17	6.61860E-03	0.71236	
18	7.39161E-03	0.76703	
19	8.32640E-03	0.87084	
20	0.01161	0.85579	
21	0.01312	0.77172	
22	0.01591	0.86923	
23	0.01755	0.83824	
24	0.02151	0.89525	
25	0.02390	0.87504	

from the first generator, then s successive values from the second one, and so on until t values have been taken. Let $L_t(s,h)$ be the lattice generated by these points, $d_t(s,h)$ and $q_t(s,h)$ the corresponding distance between hyperplanes and Beyer quotients, and $S_t(s,h) = d_t^*(m)/d_t(s,h)$. We would like the generator to perform well for s up to the largest value of t that we want to consider, and h equal to the values of $W = 2^w$ and $VW = 2^{v+w}$ that are likely to be used. We computed the values of $d_t(s,2^w)$ and $q_t(s,2^w)$ for $1 \le t \le 20$, and $S_t(s,2^w)$ for $2 \le t \le 8$, for $1 \le s \le 20$ and $30 \le w \le 80$. Table 3 gives a summary of the results. The second and third columns of the table give, for each w, the values of

$$Q^*(20, 2^w) \quad = \quad \min_{1\le t,s\le 20} q_t(s, 2^w) \tag{6}$$

and

$$S^*(8, 2^w) \quad = \quad \min_{t\le 8,\, s\le 8} S_t(s, 2^w). \tag{7}$$

Columns 4 to 7 give the indices t and s for which the minimum is attained in (6) and (7), respectively. It is interesting to see that the minima in (6) are always attained (here) for

Table 2: Lattice structure for the previous combined LCG with two components

t	d_t	q_t	S_t
2	6.50187E-10	0.50090	0.66650
3	7.00211E-07	0.70164	0.76439
4	4.63525E-05	0.14435	0.39148
5	2.00784E-04	0.59746	0.74850
6	8.89023E-04	0.61730	0.67560
7	2.62125E-03	0.61297	0.61124
8	5.78180E-03	0.57366	0.56812
9	9.57124E-03	0.55894	
10	0.01738	0.55322	
11	0.02361	0.63902	
12	0.03077	0.66354	
13	0.03477	0.67893	
14	0.03968	0.85761	
15	0.05987	0.79254	
16	0.06075	0.72478	
17	0.06509	0.70112	
18	0.07433	0.73784	
19	0.08192	0.83544	
20	0.08771	0.81133	
21	0.10206	0.84131	
22	0.10721	0.75895	
23	0.11111	0.71910	
24	0.11952	0.88423	
25	0.13131	0.89080	

small values of t and s: never more than 8 for t and never more than 7 for s. So, it seems unlikely that increasing the upper bound of 20 in (6) to some larger value, say 100, would significantly change the table.

The "best" choices for w are those for which $Q^*(20, 2^w)$ and $S^*(8, 2^w)$ are large. Small values (close to zero) of these figures of merit indicate to some extent that subsequences starting 2^w values apart in the generator's sequence are not well-behaved. For example, $w = 32$ and $w = 76$ are bad choices. How small is too small? In the present context, given the size of the modulus m, we believe that $Q^*(20, 2^w)$ near 0.1 or $S^*(8, 2^w)$ near 0.3 are acceptable.

2.5 Empirical Testing

We applied extensive statistical testing to this proposed combined generator. The tests applied included in particular the 21 tests that were applied in [4], the 10 tests applied in [6] (which many popular generators failed), plus many others. The generator passed

Table 3: Figures of merit for lacunary indices

w	$Q^*(20,w)$	$S^*(20,w)$	t_q^*	s_q^*	t_S^*	s_S^*
30	0.137171	0.241715	3	2	4	1
31	0.242620	0.362170	8	7	8	7
32	0.032311	0.135188	3	1	3	1
33	0.220591	0.383950	2	1	8	4
34	0.221913	0.389231	4	3	4	1
35	0.136449	0.312680	2	1	8	5
36	0.243299	0.337454	4	1	7	6
37	0.316370	0.325918	8	4	8	4
38	0.238970	0.480105	5	4	7	6
39	0.170184	0.286288	4	3	4	3
40	0.096302	0.308356	3	2	3	2
41	0.339710	0.453425	6	3	7	6
42	0.092104	0.362648	6	4	8	3
43	0.065889	0.290381	3	2	4	1
44	0.105515	0.302422	2	1	2	1
45	0.170428	0.344029	3	2	3	2
46	0.109773	0.308390	2	1	2	1
47	0.155933	0.337623	4	1	4	1
48	0.178559	0.301876	6	2	5	2
49	0.278264	0.419394	4	3	4	3
50	0.238707	0.319469	5	1	5	1
51	0.159467	0.246888	5	2	5	2
52	0.307683	0.394198	6	2	6	2
53	0.083400	0.268777	2	1	2	1
54	0.270062	0.365564	2	1	5	2
55	0.270539	0.348522	3	2	8	2
56	0.101693	0.228934	4	2	4	2
57	0.122271	0.338235	5	1	5	1
58	0.309237	0.401068	4	1	8	7
59	0.193495	0.249822	5	1	5	1
60	0.176330	0.313239	6	2	6	2
61	0.133205	0.368244	4	2	7	3
62	0.239344	0.334832	3	2	3	2
63	0.128482	0.345292	4	2	4	2
64	0.151889	0.366691	3	1	3	1
65	0.183404	0.227093	5	3	5	3
66	0.175168	0.243849	4	1	4	1
67	0.158625	0.382347	5	4	5	4
68	0.208553	0.364044	6	1	7	2
69	0.173252	0.350050	4	2	4	2
70	0.172407	0.297988	4	3	7	3
71	0.066515	0.285245	3	2	3	2
72	0.275884	0.461439	6	5	6	5
73	0.225032	0.388896	5	2	3	1
74	0.114698	0.298228	4	1	7	2
75	0.187393	0.407324	3	2	5	2
76	0.040866	0.136882	2	1	3	1
77	0.240086	0.347404	3	2	7	1
78	0.068412	0.322411	3	1	5	1
79	0.206676	0.365824	3	2	4	2
80	0.145853	0.323621	4	2	4	2

all these tests without any problem. So, empirically, it behaves as expected. Of course, empirical statistical testing never *proves* that a generator is safe, but at least it improves one's confidence.

3 Functional Definition of the Package

We now give the functional definition of the package. In contrast to L'Ecuyer and Côté [9], here the values of V and W can be changed by calling the procedure `Init`. The default values are $V = 2^{31}$ and $W = 2^{41}$. The initial seed can also be changed by calling `SetInitialSeed` after calling `Init`. The default initial seed is the vector $(11111111, 22222222, 33333333, 44444444)$. The generators are numbered from 0 to 100 (the latter can be increased as needed). Each generator g has an *initial seed* I_g, a *last seed* L_g (the starting point of its current segment), and a *current seed* C_g (the current state of the generator). When a random number is produced by this generator, C_g is advanced to the next state in the generator's sequence. The procedure `InitGenerator` permits one to jump back and forth between the different segments of any given generator.

```
#ifndef H_CLCG4_H
#define H_CLCG4_H

#define Maxgen  100
```

Maximum number of generators, plus one. This value can be increased as needed.

```
typedef  unsigned short int Gen;
```

A generator number, in the range 0 to `Maxgen`.

```
typedef  enum {InitialSeed, LastSeed, NewSeed}  SeedType;
```

```
void Init (long v, long w);
```

Initializes the random number package. The values of V and W will be 2^v and 2^w, respectively. It is recommended that $v \geq 30$, $w \geq 40$, and $v + w \leq 100$. The initial seed is set by default to $(11111111, 22222222, 33333333, 44444444)$.

```
void InitDefault ();
```

Initializes the package as in `Init`, with the default values $V = 2^{31}$ and $W = 2^{41}$.

```
void SetInitialSeed (long s[4]);
```

Set the initial seed I_0 of generator number 0 ($g = 0$) to the integer values `s[0],...,s[3]`. Those values must satisfy: $1 \leq \mathtt{s}[0] \leq 2147483646$, $1 \leq \mathtt{s}[1] \leq 2147483542$, $1 \leq \mathtt{s}[2] \leq 2147483422$,

$1 \leq$ s[3] ≤ 2147483322. The initial seeds of all other generators are recomputed accordingly, so they are spaced VW values apart, and all generators are reinitialized to their initial seeds.

```
void InitGenerator (Gen g, SeedType Where);
```

Reinitialize the generator g. According to the value of Where, that generator's state C_g will be reset to the initial seed I_g (InitialSeed), or to the last seed L_g (LastSeed), which is at the beginning of the current segment, or to a new seed (NewSeed), W values ahead of the last seed in the generator's sequence. The last seed L_g is also reset to the same value as C_g.

```
void SetSeed (Gen g, long s[4]);
```

Set initial seed I_g of generator g to s[1],...,s[4]. Those values must satisfy: $1 \leq$ s[0] ≤ 2147483646, $1 \leq$ s[1] ≤ 2147483542, $1 \leq$ s[2] ≤ 2147483422, $1 \leq$ s[3] ≤ 2147483322. The current state C_g and last seed L_g are also put to that seed.

```
void GetState (Gen g, long s[4]);
```

Returns the current state C_g of generator g in s[0],...,s[3].

```
void WriteState (Gen g);
```

Writes the current state C_g of generator g.

```
double GenVal (Gen g);
```

Returns a "uniform" random number over $[0, 1]$, using generator g. The current state C_g is changed, but not I_g and L_g.

```
#endif
```

---

4 Implementation in C

```
#include <stdio.h>
#include "clcg4.h"

/*********************************************************************/
/* Private part.                                                     */
/*********************************************************************/

#define H   32768               /* = 2^15 : use in MultModM.          */

static long aw[4], avw[4],      /*   a[j]^{2^w} et a[j]^{2^{v+w}}.    */
            a[4] = { 45991, 207707, 138556, 49689 },
            m[4] = { 2147483647, 2147483543, 2147483423, 2147483323 };

static long Ig[4][Maxgen+1], Lg[4][Maxgen+1], Cg[4][Maxgen+1];
                  /* Initial seed, previous seed, and current seed. */

static short i, j;

static long MultModM (long s, long t, long M)
```

```
   /* Returns (s*t) MOD M.  Assumes that -M < s < M and -M < t < M.    */
   /* See L'Ecuyer and Cote (1991).                                   */
  {
  long R, S0, S1, q, qh, rh, k;

  if (s < 0)  s += M;
  if (t < 0)  t += M;
  if (s < H)  { S0 = s;  R = 0; }
  else
    {
    S1 = s/H;  S0 = s - H*S1;
    qh = M/H;  rh = M - H*qh;
    if (S1 >= H)
      {
      S1 -= H;   k = t/qh;   R = H * (t - k*qh) - k*rh;
      while (R < 0)  R += M;
      }
    else R = 0;
    if (S1 != 0)
      {
      q = M/S1;   k = t/q;   R -= k * (M - S1*q);
      if (R > 0)  R -= M;
      R += S1*(t - k*q);
      while (R < 0)  R += M;
      }
    k = R/qh;   R = H * (R - k*qh) - k*rh;
    while (R < 0) R += M;
    }
  if (S0 != 0)
    {
    q = M/S0;   k = t/q;   R -= k* (M - S0*q);
    if (R > 0)  R -= M;
    R += S0 * (t - k*q);
    while (R < 0)  R += M;
    }
  return R;
  }

/*-------------------------------------------------------------------*/
/* Public part.                                                      */
/*-------------------------------------------------------------------*/

void SetSeed (Gen g, long s[4])
  {
  if (g > Maxgen) printf ("ERROR: SetSeed with g > Maxgen \n");
  for (j = 0; j < 4; j++)  Ig [j][g] = s [j];
  InitGenerator (g, InitialSeed);
  }

void WriteState (Gen g)
  {
  printf ("\n State of generator g = %u :", g);
  for (j = 0; j < 4; j++)
    {
    printf ("\n   Cg[%u] = %lu", j, Cg[j][g]);
    }
  printf ("\n");
  }

void GetState (Gen g, long s[4])
  {
  for (j = 0; j < 4; j++)  s [j] = Cg [j][g];
  }
```

```
void InitGenerator (Gen g, SeedType Where)
  {
  if (g > Maxgen) printf ("ERROR: InitGenerator with g > Maxgen \n");
  for (j = 0; j < 4; j++)
    {
    switch (Where)
      {
      case InitialSeed :
        Lg [j][g] = Ig [j][g];   break;
      case NewSeed :
        Lg [j][g] = MultModM (aw [j], Lg [j][g], m [j]);   break;
      case LastSeed :
        break;
      }
    Cg [j][g] = Lg [j][g];
    }
  }

void SetInitialSeed (long s[4])
  {
  Gen g;
  for (j = 0; j < 4; j++)  Ig [j][0] = s [j];
  InitGenerator (0, InitialSeed);
  for (g = 1; g <= Maxgen; g++)
    {
    for (j = 0; j < 4; j++)
      Ig [j][g] = MultModM (avw [j], Ig [j][g-1], m [j]);
    InitGenerator (g, InitialSeed);
    }
  }

void Init (long v, long w)
  {
  long sd[4] = {11111111, 22222222, 33333333, 44444444};
  for (j = 0; j < 4; j++)
    {
    aw [j] = a [j];
    for (i = 1; i <= w; i++)
      aw [j]  = MultModM (aw [j], aw [j], m[j]);
    avw [j] = aw [j];
    for (i = 1; i <= v; i++)
      avw [j] = MultModM (avw [j], avw [j], m[j]);
    }
  SetInitialSeed (sd);
  }

double GenVal (Gen g)
  {
  long k,s;
  double u;
  u = 0.0;
  if (g > Maxgen) printf ("ERROR: Genval with g > Maxgen \n");

  s = Cg [0][g];  k = s / 46693;
  s = 45991 * (s - k * 46693) - k * 25884;
  if (s < 0) s = s + 2147483647;  Cg [0][g] = s;
  u = u + 4.65661287524579692e-10 * s;

  s = Cg [1][g];  k = s / 10339;
  s = 207707 * (s - k * 10339) - k * 870;
  if (s < 0) s = s + 2147483543;  Cg [1][g] = s;
  u = u - 4.65661310075985993e-10 * s;
```

```
  if (u < 0) u = u + 1.0;

  s = Cg [2][g];  k = s / 15499;
  s = 138556 * (s - k * 15499) - k * 3979;
  if (s < 0) s = s + 2147483423;  Cg [2][g] = s;
  u = u + 4.65661336096842131e-10 * s;
  if (u >= 1.0) u = u - 1.0;

  s = Cg [3][g];  k = s / 43218;
  s = 49689 * (s - k * 43218) - k * 24121;
  if (s < 0) s = s + 2147483323;  Cg [3][g] = s;
  u = u - 4.65661357780891134e-10 * s;
  if (u < 0) u = u + 1.0;

  return (u);
  }

void InitDefault ()
  {
  Init (31, 41);
  }
```

Acknowledgments

This work has been supported by NSERC-Canada grants # ODGP0110050 and # SMF0169893, FCAR-Québec grant # 93ER1654, and the Canadian Nuclear Fuel Waste Management Program, which is jointly funded by Atomic Energy of Canada Limited and Ontario Hydro under the auspices of the CANDU Owners Group. Anna Bragina helped implementing the package. Martin Boivin and Hannes Leeb helped testing it.

References

[1] P. Bratley, B. L. Fox, and L. E. Schrage. *A Guide to Simulation.* Springer-Verlag, New York, second edition, 1987.

[2] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Addison-Wesley, Reading, Mass., second edition, 1981.

[3] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis.* McGraw-Hill, New York, second edition, 1991.

[4] P. L'Ecuyer. Efficient and portable combined random number generators. *Communications of the ACM*, 31(6):742–749 and 774, 1988. See also the correspondance in the same journal, 32, 8 (1989) 1019–1024.

[5] P. L'Ecuyer. Random numbers for simulation. *Communications of the ACM*, 33(10):85–97, 1990.

[6] P. L'Ecuyer. Testing random number generators. In *Proceedings of the 1992 Winter Simulation Conference*, pages 305–313. IEEE Press, Dec 1992.

[7] P. L'Ecuyer. Uniform random number generation. *Annals of Operations Research*, 53:77–120, 1994.

[8] P. L'Ecuyer. Random number generation. In Jerry Banks, editor, *Handbook on Simulation*. Wiley, 1997. To appear. Also GERAD technical report number G-96-38.

[9] P. L'Ecuyer and S. Côté. Implementing a random number package with splitting facilities. *ACM Transactions on Mathematical Software*, 17(1):98–111, 1991.

[10] P. L'Ecuyer and R. Couture. An implementation of the lattice and spectral tests for multiple recursive linear random number generators. *INFORMS Journal on Computing*, Circa 1997. To appear.

[11] P. L'Ecuyer and S. Tezuka. Structural properties for two classes of combined random number generators. *Mathematics of Computation*, 57(196):735–746, 1991.

[12] H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*, volume 63 of *SIAM CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM, Philadelphia, 1992.

[13] B. A. Wichmann and I. D. Hill. An efficient and portable pseudo-random number generator. *Applied Statistics*, 31:188–190, 1982. See also corrections and remarks in the same journal by Wichmann and Hill, **33** (1984) 123; McLeod **34** (1985) 198–200; Zeisel **35** (1986) 89.