Implementing a Random Number Package with Splitting Facilities

PIERRE L'ECUYER and SERGE CÔTÉ
Laval University

Multiple generators are often required in simulation studies, for instance, to facilitate synchronization for variance reduction purposes, and multiple independent streams per generator are helpful to make independent replications.

A portable set of software tools is described for uniform random variates generation. It provides for multiple generators running simultaneously, and each generator has its sequence of numbers partitioned into many long (disjoint) substreams. Simple procedure calls allow the user to make any generator "jump" ahead to the beginning of its next substream, back to the beginning of its current substream, or back to the beginning of its first substream. A simple switch permits a change from regular to antithetic variates or vice versa. Implementation issues are discussed An efficient and portable code is also provided for computing (as MOD m) for any positive integer values of $a < m$, $s < m$, and $m < 2^{b-1}$ on a b-bit computer. This code is used to implement the package.

A Pascal implementation for 32-bit computers is described. The basic underlying generator for this implementation has been proposed in a previous paper; it combines two multiplicative linear congruential generators and has a period of $p \approx 2\ 3 \times 10^{18}$

Categories and Subject Descriptors: G.3 [**Mathematics of Computing**]: Probability and Statistics—*random number generation, statistical software*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Combined generators, disjoint streams, portability, repeatability

1. INTRODUCTION

1.1 Basic Generators

A facility for generating sequences of pseudorandom numbers is a fundamental part of computer simulation systems. Usually, in practice, such a facility produces a deterministic sequence of values, but externally these values should *appear* to be drawn independently from a uniform distribution between 0 and 1 [1, 6].

This research was supported by the Canadian Natural Sciences and Engineering Research Council under grant A5463 and by Québec's "Fonds pour la Formation de Chercheurs et l'Aide à la Recherche" under grant EQ2831.

Author's address: Département d'IRO, Univeristé de Montréal, C.P. 6128, Succ. A, Montéal, H36 3J7, Canada.

A commonly employed kind of generator is the multiplicative linear congruential generator (MLCG), which goes from integer to integer according to the recursion

$$s_i = as_{i-1} \text{MOD } m \tag{1}$$

where the modulus m and the multiplier $a < m$ are positive integers. Each integer s_i takes its values from the set of states $S = \{1, 2, \ldots, m-1\}$, and the sequence is periodic with period $p \leq m - 1$. A simple mapping may convert s_i to a floating point value between 0 and 1. If m is prime and a is a primitive element modulo m, then the MLCG has maximal period ($p = m - 1$) [6, p. 19]. For practical considerations (ease of implementation), one usually chooses m small enough to be representable as a regular integer on the target machine and a such that $a^2 < m$ (see [1, 8]).

To increase the period and improve the empirical statistical behavior without increasing m, one could run two or more MLCGs in parallel and combine their states, producing a sequence whose period is the least common multiple of the individual periods. Consider a family of l maximal period MLCGs where for $j = 1, \ldots, l$, generator j has modulus m_j and multiplier a_j:

$$s_{j,i} = a_j s_{j,i-1} \text{MOD } m_j. \tag{2}$$

For a given initial state $s_0 = (s_{1,0}, \ldots, s_{l,0})$ (called the *seed*), the period p of the sequence $\{s_i = (s_{1,i}, \ldots, s_{l,i}),\ i = 0, 1, 2, \ldots\}$ is the least common multiple of $m_1 - 1, \ldots, m_l - 1$ (see [8, Lemma 2]). L'Ecuyer [8] proposes the combination:

$$Z_i = \left(\sum_{j=1}^{l} (-1)^{j-1} s_{j,i} \right) \text{MOD}(m_1 - 1) \tag{3}$$

which yields an approximation of a uniform discrete random variable on $\{0, \ldots, m_1 - 2\}$. It could be transformed into $U_i \in (0, 1)$ by:

$$U_i = \begin{cases} Z_i / m_1 & \text{if } Z_i > 0 \\ (m_1 - 1)/m_1 & \text{if } Z_i = 0. \end{cases} \tag{4}$$

Two generators, easy to code in a high level language, have been proposed by L'Ecuyer [8]. The first one uses $l = 2$ MLCGs, has period $p \approx 2.3 \times 10^{18}$, and is for 32-bit computers; the other one uses $l = 3$ MLCGs, has period $p \approx 8.1 \times 10^{12}$, and is for 16-bit computers.

1.2 The Need for Multiple Substreams

Many disjoint random number subsequences are often required in simulation studies, for instance, (1) to make independent replications or/and (2) to associate distinct "virtual" generators with different sources of randomness in the system to facilitate synchronization for variance reduction [1]. When both cases apply, one needs many subsequences for each virtual generator, so that the full sequence will be divided into disjoint substreams, and each

substream subdivided into disjoint subsubstreams. To produce such a "splitting," different seeds (values of $\{s_i = (s_{1,i}, \ldots, s_{l,i})\}$ must be obtained far enough apart in the sequence to insure that the substreams do not overlap.

In other words, given any seed s_i and positive integer j, there should be a quick way to compute s_{i+j} (without generating all intermediate values, of course). In principle, that can be done quite easily for an MLCG, since

$$s_{i+j} = (a^j s_i) \text{MOD } m = (a^j \text{MOD } m) s_i \text{MOD } m. \tag{5}$$

For any given j, $(a^j \text{MOD } m)$ can be precomputed.

Most packages offer no facility for jumping ahead directly from s_i to s_{i+j} or to compute distant seeds efficiently (see [3, 5, 13, 14], for instance). Some provide specific seeds to generate disjoint streams [1] and many simulation languages offer a limited number of virtual generators (usually no more than 10), all based on the same generator, but using fixed starting seeds set say 100,000 values apart (see [2, 3, 12]). This provides relatively low flexibility. Suppose, for instance, that you want to perform independent pairs of replications with common random numbers across the configurations (i.e., between any two runs of the same pair) in order to compare two different configurations of a system. To insure proper synchronization, you want every generator to start from the same seed in both runs of the same pair. However, in general, these two runs will make a different number of calls to a generator, and programming "tricks" should be used to skip a proper amount of random numbers to resynchronize the generators for the next pair without overlap in the random number streams [1, Sec. 8.2] and [7, Sec. 11.2]. This requires extra programming effort and could be error prone. Good software tools should ease the programmer's task in that respect. A simple procedure call should permit resetting a generator to a previous seed or jumping ahead to a new seed for the next run. Of course, the sequence of "new seeds" (one per run) should be the same for both configurations of the system. Implementing such tools requires efficient "jumping ahead" facilities, which in turn ask for efficient procedures to compute as MOD m.

1.3 Overview

In Section 2 of this paper, we propose and compare three general methods for computing as MOD m for any positive integer values of $a < m$, $s < m$, and $m < 2^{b-1}$ using only integers between -2^{b-1} and 2^{b-1} (strictly), where b is a positive integer. Thus, such computation can be done on a b-bit computer whenever m is representable on that computer. These techniques are interesting and useful for other applications as well, for example, for implementing the spectral test [6].

Section 3 describes a small package with multiple virtual generators (associated with disjoint substreams) and multiple subsubstreams per generator. It can be read independently of Section 2. We provide a Pascal implementation for computers with 32 bits or more. Implementations for computers having smaller word sizes, or using more than 32 bits, can be done in a similar way.

2. COMPUTING (as MOD m)

Consider a b-bit computer on which all integers between -2^{b-1} and 2^{b-1} (exclusive) are well represented. We want to compute (as MOD m), where a, s, and m are positive integers smaller than 2^{b-1}. Without loss of generality, we assume that $a < m$ and $s < m$ (if not, replace a and s by a MOD m and s MOD m, respectively). We also assume that b is even, which is the case on all conventional computers, and let $d = (b - 2)/2$.

When m is a power of 2, this is quite simple: multiply as without checking for overflow and discard the higher order bits. But even for this simple case, writing a portable code in a high-level language not designed for bit operations is not trivial. For general m, this is still more tricky. Performing the computations in DOUBLE PRECISION is not a solution, since DOUBLE PRECISION variables usually have much less than $2^{2(b-1)}$ bits of precision. On a VAX computer (32 bit), for instance, they carry no more than 55 bits of accuracy, while the product as can exceed 2^{61}. A method is described by Bratley et al. [1] and Payne et al. [11] for the case where m is just a little smaller than a power of 2. It uses only integers, but works only under certain conditions on m and a.

2.1 Approximate Factoring

A more efficient method is proposed in [1, Sec. 6.5.2], for the case where

$$a^2 \le m. \tag{6}$$

It operates as follows. Define $q = \lfloor m/a \rfloor$ and $r = m$ MOD a. Then (see [1, 8]):

$$as \text{ MOD } m = \big(a(s \text{ MOD } q) - \lfloor s/q \rfloor r\big) \text{MOD } m. \tag{7}$$

When computing the right-hand side of (7), it is easily seen [1, 8] that all intermediate values remain in the range from $-m$ to m (inclusive). We call this method approximate factoring (AF). Notice that the constraint $a^2 \le m$ cannot be replaced by the less restrictive constraint $a^2 < 2^{b-1}$, since $\lfloor s/q \rfloor r$ could overflow if q is too small. When $a^2 < 2^{b-1}$, one can only show that $\lfloor s/q \rfloor r < 2^b$. However, when $a^2 \le 2^{b-2}$ (i.e., $a \le 2^d$), then $\lfloor s/q \rfloor r < \lfloor (aq + r)/q \rfloor r < 2a^2 \le 2^{b-1}$ and overflow never occurs, so that AF could also be used. In the latter case, all intermediate values in the following instructions remain strictly between -2^{b-1} and m:

```
k := s DIV q;
s := a*(s - k*q) - k*r;
WHILE s < 0 DO s := s + m
```

The number of turns into the WHILE loop is at most $\lceil 2^{b-1}/m \rceil$. For our applications, we usually have $m \ge 2^{b-2}$, in which case the body of the WHILE loop never executes more than twice. Notice, however, that for small m, this code could be rather inefficient in the worst case. For instance, if $a = 2^{15} - 1 = 32767$, $m = 2a - 1 = 65533$, and $s = m - 1 = 65532$, then $q = 1$, $r = a - 1 = 32766$, and the statement s := s + m must be executed 32765 times! Fortunately, such bad combinations occur very rarely in

practice, and empirical results (see Table I below) tell that this code is very efficient for the average case (random values of $a < m$ and $s < m$) even for small values of m.

2.2 Recursive Reduction

When (6) does not hold, one could sometimes decompose a as $a = a_1 \times a_2$ such that $a_1^2 < m$ and $a_2^2 < m$. The AF method is then applied twice to compute

$$as \text{ MOD } m = (a_1(a_2 s \text{ MOD } m))\text{MOD } m.$$

This approach is used by Marse and Roberts [9]. Besides the fact that factoring a is time consuming, the major limitation is that a decomposition in two factors as above does not always exist.

When $a^2 > m$, the term susceptible to overflow in (7) is $\lfloor s/q \rfloor r$. Equation (7) can be rewritten as

$$as \text{ MOD } m = (a(s \text{ MOD } q) - (\lfloor s/q \rfloor r \text{ MOD } m))\text{MOD } m. \tag{8}$$

if $r^2 \leq m$ or $r \leq 2^d$, then AF can be used to compute $\lfloor s/q \rfloor r$ MOD m. Otherwise, this term can be computed by using formula (8) recursively, after resetting $a := r$, $s := \lfloor s/q \rfloor$, $q := \lfloor m/a \rfloor$, and $r := m$ MOD a. Recursive reductions are thus performed until $a \leq 2^d$ or $r = 0$ or $\lfloor s/q \rfloor = 0$ (we avoid checking if $a \leq \sqrt{m}$ since computing the square root of m is too time consuming). Since r decreases at each recursive call (each iteration), convergence occurs after a finite number of iterations. The worst case upper bound on the number of iterations is (approximately) $\sqrt{m}$, which is rather high, but convergence occurs rather quickly in the average case (random choice of a, s, and m). For instance, for $b = 32$, m near 2^{31}, and a and s uniformly distributed over $\{1, \ldots, m-1\}$, the average number of iterations is approximately between 5 and 6, according to our empirical investigations. This *recursive reduction* approach is completely general. A Pascal code appears in Figure 1. Partial sums are kept in variable p. To avoid overflow when adding the next term to p, the constant m can be added or subtracted without affecting the result, since all computations are done modulo m.

2.3 Decomposition

A more direct approach, based on *decomposition*, operates as follows. Rewrite a as

$$a = a_0 + a_1 2^d + a_2 2^{2d} \tag{9}$$

where $0 \leq a_0 < 2^d$, $0 \leq a_1 < 2^d$, and $a_2 = 0$ or 1. Then

$$\begin{aligned} as \text{ MOD } m &= (a_0 s)\text{MOD } m \\ &\quad + ((a_1 s)\text{MOD } m)2^d \text{MOD } m \\ &\quad + ((a_2 s)\text{MOD } m)2^{2d} \text{MOD } m \\ &= ((((a_2 2^d s)\text{MOD } m + (a_1 s)\text{MOD } m)\text{MOD } m)2^d \text{MOD } m \\ &\quad + a_0 s \text{ MOD } m)\text{MOD } m. \end{aligned} \tag{10}$$

```
FUNCTION MultMod_Reduc (a, s, m: INTEGER): INTEGER;
  {Assumes 0 < a < m and 0 < s < m  Returns (a*s)MOD m.}
  CONST
    H = 32768;      {Should be the value of 2**d}
  VAR
    q, r, k, p, Sign: INTEGER;
  BEGIN
  q := m DIV a;   r := m - a*q;   k := s DIV q;
  p := a*(s - k*q);   Sign := -1;
  WHILE (a > H) AND (r < > 0) AND (k < > 0) DO
    BEGIN
    a := r;   s := k;
    q := m DIV a;   r := m - a*q;   k := s DIV q;
    p := p + Sign*a*(s - k*q);   Sign := -Sign;
    IF Sign*p > 0 THEN p := p - Sign*m
    END;
  IF (k < > 0) THEN p := p + Sign*k*r;
  WHILE p < 0 DO p := p + m;
  MultMod_Reduc := p
  END
```

Fig. 1. Iterative Implementation of recursive reduction.

In each of the four products modulo m in (10), one of the factors is smaller or equal to 2^d, so AF can be applied. Figure 2 shows a Pascal FUNCTION implementing this method. It runs in $O(1)$ time, except for the WHILE loops, and as said previously, the number of turns into each WHILE loop is at most $\lceil 2^{b-1}/m \rceil$. For our applications, we usually have $m \geq 2^{b-2}$, and this means at most two turns.

2.4 Russian Peasant Method

A third general approach is based on the *Russian peasant* method [6]: to multiply integers x and y, initialize a counter to zero, halve x and double y iteratively until $x = 0$, adding y to the counter every time x is odd. The final value of the counter holds the product. The variation here is that doubling is done modulo m. An implementation appears in Figure 3. As soon as $a \leq 2^d$, we stop iterative halving and doubling to apply AF. The number of turns into the first WHILE loop is never more than d. For the second one, it is never more than 2 when $m \geq 2^{b-2}$. For applications where very few different values of m are used, and especially for small values of m, it might be profitable to precompute sm = $\sqrt{m}$, and replace the condition (a > H) by (a > sm), and the second WHILE . . . DO by an IF . . . THEN.

2.5 Empirical Comparisons

We performed an empirical speed comparison of these techniques on a VAX-11/780 computer (on which $b = 32$), using VAX-11 Pascal version 3.6 [16] under VAX/VMS version 4.6, and on an IBM-PS2/50 and an IBM-PC/XT, both with $b = 16$ and using Borland's Turbo-Pascal version 3.0 [15]. We also included in our comparison an implementation of AF for the case where $a^2 \leq m$. For each case, we performed two runs, choosing randomly 100 values

```
FUNCTION MultMod_Decompos (a, s, m: INTEGER): INTEGER;
  {Assumes 0 < a < m and 0 < s < m. Returns (a*s)MOD m }
  CONST
    H = 32768;      {Should be the value of 2**d}
  VAR
    a0, a1, q, qh, rh, k, p: INTEGER,
  BEGIN
  IF a < H THEN
    BEGIN a0 := a;   p := 0 END
  ELSE
    BEGIN
    a1 := a DIV H;   a0 := a - H*a1;
    qh := m DIV H;   rh := m - H*qh;
    IF a1 >= H THEN {a2 = 1}
      BEGIN
      a1 := a1 - H,   k := s DIV qh;
      p := H*(s - k*qh) - k*rh;
      WHILE p < 0 DO p := p + m
      END
    ELSE
      p := 0;
    {p = (a2*s*h)MOD m}
    IF a1 < > 0 THEN
      BEGIN
      q := m DIV a1;   k := s DIV q,
      p := p - k*(m - a1*q);   IF p > 0 THEN p := p - m;
      p := p + a1*(s - k*q);   WHILE p < 0 DO p := p + m
      END;
    {p = ((a2*h + a1)*s)MOD m}
    k := p DIV qh;   p := H*(p - k*qh) - k*rh;
    WHILE p < 0 DO p := p + m
    END;
  {p = ((a2*h + a1)*h*s)MOD m}
  IF a0 < > 0 THEN
    BEGIN
    q := m DIV a0;   k := s DIV q;
    p := p - k*(m - a0*q);   IF p > 0 THEN p := p - m;
    p := p + a0*(s - k*q);   WHILE p < 0 DO p := p + m
    END;
  MultMod_Decompos := p
END
```

Fig. 2. Pascal implementation of decomposition method.

of m in the set $\{1, \ldots, 2^{b-1} - 1\}$ for the first run and 100 values of m in the set $\{2^{b-2}, \ldots, 2^{b-1} - 1\}$ for the second run. For each value of m, 100 values of a and s were drawn at random from the set $\{1, \ldots, m - 1\}$ (from $\{1, \ldots, \lfloor \sqrt{m} \rfloor\}$ in the case of AF). For each method, the average CPU time per function call was computed. To obtain confidence intervals, the whole experiment was repeated 10 times, using different seeds to generate the random values. Very low variations were observed across repetitions. Table I gives the results, in the form of 90-percent confidence intervals. These values include the times for function calling and parameter passing, which is about

```
FUNCTION MultMod_Russian (a, s, m: INTEGER): INTEGER;
  {Assumes 0 < a < m and 0 < s < m. Returns (a*s)MOD m.}
CONST
  H = 32768;      {Should be the value of 2**d}
VAR
  q, r, k, p: INTEGER;
BEGIN
p := -m;
WHILE (a > H) DO
  BEGIN
  IF ODD (a) THEN
    BEGIN
    p := p + s;   IF p > 0 THEN p := p - m
    END,
  a := a DIV 2:
  s := (s - m) + s;   IF s < 0 THEN s := s + m
  END;
q := m DIV a;   k := s DIV q;
s := a*(s - k*q) - k*(m - q*a);
WHILE s < 0 DO s := s + m;
p := p + s;   IF p < 0 THEN p := p + m;
MultMod_Russian := p
END
```

Fig. 3. Implementation of Russian peasant approach.

Table I. Empirical Speed Comparison of Four Methods to Compute *as* MOD *m* (average time per call in *ms*)

	AF	Recursive Reduction	Decom-position	Russian Peasant
VAX/780				
$1 \le m < 2^{31}$	0.024 ± 0.003	0.264 ± 0.003	0.150 ± 0.004	0.265 ± 0.004
$2^{30} \le m < 2^{31}$	0.033 ± 0.003	0.279 ± 0.007	0.178 ± 0.005	0.290 ± 0.007
IBM-PS2/50				
$1 \le m < 2^{15}$	0.083 ± 0.002	0.243 ± 0.002	0.175 ± 0.001	0.190 ± 0.002
$2^{14} \le m < 2^{15}$	0.084 ± 0.001	0.245 ± 0.001	0.175 ± 0.001	0.192 ± 0.002
IBM-PC/XT				
$1 \le m < 2^{15}$	0.548 ± 0.001	1.89 ± 0.008	1.36 ± 0.003	1.28 ± 0.006
$2^{14} \le m < 2^{15}$	0.556 ± 0.001	2.00 ± 0.002	1.38 ± 0.002	1.38 ± 0.002

0.004 ms on the VAX, about 0.025 ms on the PS2/50, and about 0.13 ms on the PC/XT on the average. The (high) overhead induced by accessing the "clock" of the current process, generating the values of m, a, and s, loop control, and statistical collection, was estimated by running the same programs without the function calls, and removed by subtraction. The computed 90-percent confidence intervals for these values were 0.599 ± 0.003 ms for the VAX, 0.434 ± 0.001 ms for the IBM/PS2, and 3.087 ± 0.001 ms for the IBM/PC. Notice that the statistical collection and variate generation were done differently on the VAX and on the microcomputers, and this is the reason why it took more time on the VAX than the PS2. The upper bound on

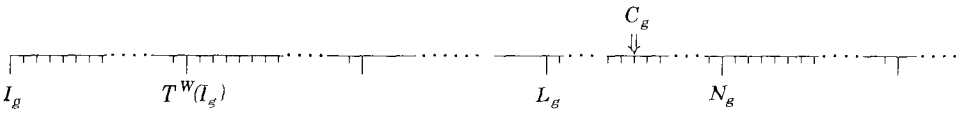
m is 2^{31} on the VAX and 2^{15} on the microcomputers. Therefore, computations were "easier" on the latter machines, and the results of Table I cannot be used to compare the relative speeds of the machines.

Of course, as usual, the results of such empirical tests are highly dependent on the computer and compiler used. Their aim is just to get a rough idea of the relative speeds in practice. On the VAX, we see that among the general methods, the fastest one is decomposition. However, it is the one with the most complex source code. When $a^2 < m$, one may use approximate factoring, which is about six times as fast. On the PS2 and PC/XT, Russian peasant, which is simpler, is about as good as decomposition. The explanation is that when b is smaller, the former goes into fewer turns into its first WHILE loop. The worst case bound on this number is linear in b. Decomposition, on the other hand, takes an $O(1)$ time when m is near 2^{b-1} and should dominate on computers with larger word sizes.

3. A RANDOM NUMBER PACKAGE

We now propose a portable set of utilities for random number generation. We consider a basic underlying generator of period p. Let s_0 be the basic seed (initial state) for this generator and $s_1, s_2, \ldots$ be its sequence of successive states. Let T denote the transition function of the generator, that is, the operator T: $S \to S$ such that $T(s_i) = s_{i+1}$, and T^k its k-fold composition ($T^k(s_i) = s_{i+k}$). Let v, w, and G be three positive integers such that $G2^v2^w \leq p$, and let $V = 2^v$ and $W = 2^w$. We partition the sequence $s_0, s_1, \ldots, s_{GVW-1}$ into G disjoint subsequences, starting from states $I_1 = s_0$, $I_2 = s_{VW} = T^{VW}(I_1)$, $I_3 = s_{2VW} = T^{VW}(I_2), \ldots, I_G = s_{(G-1)VW} = T^{VW}(I_{G-1})$, respectively. Each of these subsequences corresponds to a "virtual" generator g, for $g = 1, \ldots, G$, and it is further partitioned into V subsubsequences of length W, starting from states $I_g = s_{(g-1)VW}$, $T^W(I_g) = s_{((g-1)V+1)W}, \ldots, T^{(V-1)W}(I_g) = s_{((gV-1)W}$, respectively. I_g is called the *initial seed* of generator g.

At any moment during program execution, generator g is in some state C_g, say, in its subsubsequence number k, that is, such that $C_g = T^{(k-1)W+n}(I_g)$, where $0 \leq n < W$. We call C_g the *current state* of generator g. We also define its *last seed* $L_g = T^{(k-1)W}(I_g)$ and its *next seed* $N_g = T^{kW}(I_g) = T^W(L_g)$. L_g is the starting state of the current subsubsequence, and N_g the starting state of the next one.



In the proposed package, $s_0 = I_1$ has a fixed initial value. Global or static variables memorize the current values of C_g, I_g, and L_g for each generator g. An initialization routine computes the initial values of I_g according to the definition above and initializes C_g and L_g to I_g for $g = 1, \ldots, G$. In a simulation language or package, this routine would be called automatically at the beginning of the simulation program execution.

A procedure Init_Generator allows the user to reset the state C_g of generator g either to its initial seed I_g, or to its last seed L_g, or to a new seed which corresponds to its next seed N_g. The value of L_g is updated accordingly. Thus each generator can produce many disjoint streams (if V and W are large enough) and each stream is independent of the lengths of the previous streams. Suppose, for instance, that you want to compare two or more configurations of a system using simulation with common random numbers [1], and that you do many replications. For variance reduction purposes, you will probably use different generators for different sources of randomness in the system. Before each replication, you will also call Init_Generator to obtain a new seed for each generator in use to make sure that for every replication the same streams are used for all the configurations.

The initial seed $s_0 = I_1$ is normally set automatically, but the user can choose a different one, if he or she wishes to do so, by using the procedure Set_Initial_Seed. Upon calling this procedure, all I_g are recomputed so that $I_g = T^{VW}(I_{g-1})$ for $g = 2, \ldots, G$, and each C_g and L_g is reset to I_g.

One can also modify the initial seed of a given generator without affecting the other ones (Set_Seed) or advance its state by 2^k values for some positive integer k (Advance_State). Notice, however, that after calling one of these two procedures, the initial seeds are no longer spaced VW values apart. Procedure Get_State makes it possible to read the state E_g of generator g.

Two functions generate uniformly distributed random values: Uniform_01 generates values (strictly) between 0 and 1, while Random generates integers from the set $\{1, \ldots, N\}$, where N is a very large integer. These values could be transformed as desired for generating values following other distributions. Function Random is provided because it is sometimes preferable to use directly the integer produced by the generator instead of using its transformation into a floating point number (for instance, when generating uniform random integers over an arbitrary interval; see [1, Sec. 6.7.1]).

Each generator can also produce antithetic values (with respect to the values normally produced). The system maintains a set of Boolean flags for that purpose, one per generator, and the procedure Set_Antithetic permits to turn them on or off. If x is the value normally returned by Uniform_01 (when its flag is set to FALSE), then $1 - x$ is the antithetic value, returned when its flag is set to TRUE. The same holds for Random, with x and $1 - x$ replaced by n and $N - 1 - n$, respectively.

A user's view of the available tools appears below in Pascal notation. It corresponds to an implementation for 32-bit (or more) computers, based on the generator proposed in [8], in which $G = 32$, $V = 2^{20}$, $W = 2^{30}$, and $N = 2147483562$. An implementation code is given in the appendix. The package can also be easily implemented using a different base generator for which "jumping ahead" facilities are provided.

```
CONST
  Maxg = 32;
TYPE
  Gen = 1..Maxg;      {A generator number.}
  Seed_Type = (Initial_Seed, Last_Seed, New_Seed);
```

PROCEDURE Init_Generator (g: Gen; Where: Seed_Type);

Reinitializes the state of generator g to its initial seed I_g, or to its last seed L_g, or a new seed ($= N_g$) set 2^w values apart form L_g in the basic sequence.

PROCEDURE Set_Initial_Seed (S1, S2 INTEGER);

Resets the initial seed of generator 1 to the values S1 and S2, which must satisfy: $1 \le$ S1 ≤ 2147483562 and $1 \le$ S2 ≤ 2147483398. The initial seeds of all other generators are recomputed accordingly, and all generator's states are reset to these initial seeds. This procedure is called automatically at the beginning of program execution, with (default) parameter values S1 = 1234567890 and S2 = 123456789.

PROCEDURE Set_Seed (g: Gen; S1, S2: INTEGER);

Resets the initial seed and the state of generator g to the (S1, S2), which must satisfy $1 \le$ S1 ≤ 2147483562 and $1 \le$ S2 ≤ 2147483398. The states and seeds of other generators remain unchanged.

PROCEDURE Advance_State (g: Gen; k. INTEGER);

Advances the state of generator g by 2^k values and resets its initial seed to that value.

PROCEDURE Get_State (g: Gen; VAR S1, S2. INTEGER);

Returns in (S1, S2) the state E_g of generator g.

PROCEDURE Set_Antithetic (g: Gen; B: BOOLEAN),

When B is set to TRUE, generator g starts generating antithetic values, until it is reset to FALSE. Initially, all generators produce nonantithetic values.

FUNCTION Random (g. Gen). INTEGER;

Returns a random integer following a uniform distribution over $\{1, \ldots, 2147483562\}$, using generator number g.

FUNCTION Uniform_01 (g. Gen). REAL;

Returns a random value following the uniform distribution between 0 and 1 (strictly), using generator number g.

APPENDIX. A PASCAL CODE FOR 32-BIT (OR MORE) COMPUTERS

In the code below, the user may modify the values of Maxg, v, and w, but we recommend that v + w + lg(Maxg) ≤ 60, where lg denotes the log in base 2. If these values are modified, the constants a1_W, a2_W, a1_VW, and a2_VW must be recomputed accordingly. An efficient way to precompute $a^{2^j} \text{MOD } m$ is to start with a and square it j times modulo m, using the function MultMod. The given implementation of Uniform_01 simply multiplies a random integer by 1/m1. Alternative (more involved) implementations are suggested and justified in [9, 10].

```
  ...
Maxg = 32;
CONSTANT
  {v = 20, w = 30;}
  m1    = 2147483563;  m2    = 2147483399;
  a1    = 40014;       a2    = 40692,
  a1_W  = 1033780774;    {= a1**(2**w)MOD m1      }
  a2_W  = 1494757890;    {= a2**(2**w)MOD m2      }
  a1_VW = 2082007225;    {= a1**(2**(v + w))MOD m1 }
  a2_VW = 784306273;     {= a2**(2**(v + w))MOD m2 }
```

```
TYPE
  Gen = 1..Maxg;                {A generator number.          }
  Seed_Type = (Initial-Seed, Last_Seed, New_Seed);
VAR
  Ig_1, Ig_2, Lg_1, Lg_2, Cg_1, Cg_2: ARRAY [Gen] OF INTEGER;
  Antithetic: ARRAY [Gen] OF BOOLEAN;
FUNCTION MultMod (a, s, m: INTEGER): INTEGER;
  {Same as MultMod_Decompos in figure 2.}
  ...
PROCEDURE Init_Generator (g: Gen; Where: Seed_Type);
  BEGIN
  CASE Where OF
    Initial_Seed:
      BEGIN
      Lg_1 [g] := Ig_1 [g];   Lg_2 [g] .= Ig_2 [g]
      END;
    Last_Seed: ;
    New_Seed:
      BEGIN
      Lg_1 [g] := MultMod (a1_W, Lg_1 [g], m1);
      Lg_2 [g] := MultMod (a2_W, Lg_2 [g], m2)
      END
    END;
  Cg_1 [g] := Lg_1 [g];   Cg_2 [g] := Lg_2 [g]
  END;
PROCEDURE Set_Initial_Seed (S1, S2: INTEGER);
  VAR
    g: INTEGER;
  BEGIN
    Ig_1 [1] := S1;   Ig_2 [1] := S2;
  Init_Generator (1, Initial_Seed);
  FOR g := 2 TO Maxg DO
    BEGIN
    Ig_1 [g] := MultMod (a1_VW, Ig_1 [g - 1], m1);
    Ig_2 [g] := MultMod (a2_VW, Ig_2 [g - 1], m2);
    Init_Generator (g, Initial_Seed)
    END
  END;
PROCEDURE Set_Seed (g: Gen; S1, S2: INTEGER);
  BEGIN
  Ig1 [g] := S1;   Ig2 [g] := S2;    Init_Generator (g, Initial_Seed)
  END;
PROCEDURE Advance_State (g: Gen, k: INTEGER);
  VAR
    B1, B2, I: INTEGER;
  BEGIN
    B1 := a1; B2 := a2;
  FOR I := 1 TO k DO
    BEGIN
    B1 := MultMod (B1, B1, m1);
    B2 := MultMod (B2, B2, m2)
    END;
  {B1 = a1**k and B2 = a2**k.      }
  Set_Seed (g, MultMod (B1, Cg1 [g], m1), MultMod (B2, Cg2 [g], m2))
  END;
```

```
PROCEDURE Get_State (g: Gen; VAR S1, S2· INTEGER),
BEGIN
S1 := Cg1 [g]; S2 := Cg2 [g]
END;

PROCEDURE Set_Antithetic (g: Gen, B· BOOLEAN);
  BEGIN
  Antithetic [g] ·= B
END;

FUNCTION Random (g. Gen): INTEGER;
  VAR
    Z, k, s1, s2 INTEGER,
  BEGIN
  s1 ·= Cg_1 [g], s2 .= Cg_2 [g];
  k := s1 DIV 53668,
  s1 ·= 40014*(s1 − k*53668) − k*12211;
  IF s1 < 0 THEN s1 .= s1 + 2147483563;
  k := s2 DIV 52774,
  s2 := 40692*(s2 − k*52774) − k*3791;
  IF s2 < 0 THEN s2 .= s2 + 2147483399,
  Cg_1 [g] .= s1; Cg_2 [g] := s2;
  Z := s1 − s2;
  IF Z < 1 THEN Z = Z + 2147483562;
  IF Antithetic [g] THEN Z .= 2147483563 − Z,
  Random := Z
  END;

FUNCTION Uniform_01 (g: Gen): REAL;
  BEGIN
  Uniform_01 ·= Random (g)*4 656613057E-10
  END;

PROCEDURE Initialize;
  VAR
    g· INTEGER,
  BEGIN
    FOR g .= TO Maxg DO Antithetic [g] := FALSE;
  Set_Initial_Seed (1234567890, 123456789)
  END,

BEGIN {Main program}
Initialize
  ...
END
```

REFERENCES

1. Bratley, P., Fox, B. L., and Schrage, L. E. *A Guide to Simulation*, 2nd ed. Springer-Verlag, New York, 1987.
2. Braun, J. E. SIMSCRIPT II.5 Reference Handbook, 2nd ed., C.A.C.I., Los Angeles, Calif., 1983.
3. Chandrasekaran, U., and Sheppard, S. Implementation and analysis of random variate generators in Ada. *J. Pascal, Ada & Modula-2 5*, 4 (July/Aug. 1986), 27-39.
4. Dudewicz, E. J., Karian, Z. A., and Marshall III, R. J. Random number generation on microcomputers. In *Modeling and Simulation on Microcomputers: 1985*. The Society for Computer Simulation, 1985, pp. 9-14.
5. IMSL Library User's Manual, Edition 9.2, IMSL Inc., Houston, Tex., 1984.

6. KNUTH, D. E. *The Art of Computer Programming: Seminumerical Algorithms*, Vol. 2, 2nd ed., Addison-Wesley, Reading, Mass., 1981.
7. LAW, A. M., AND KELTON, W. D. *Simulation Modeling and Analysis*, McGraw-Hill, New York, 1982.
8. L'ECUYER, P. Efficient and portable combined random number generators. *Commun. ACM 31*, 6 (June 1988), 742–749, 774.
9. MARSE, K., AND ROBERTS, S. D. Implementing a portable FORTRAN uniform (0, 1) generator. *Simulation 41*, 4 (Oct. 1983), 135–139.
10. MONAHAN, J. F. Accuracy in random number generation. *Math. Computation 45*, 172 (Oct. 1985), 559–568.
11. PAYNE, W. H., RABUNG, J. R., AND BOGYO, T. P. Coding the Lehmer pseudo-random number generator. *Commun. ACM 12*, 2 (Feb. 1969), 85–86.
12. PEGDEN, C. D. Introduction to SIMAN, Version 3.0, Systems Modeling Corporation, State College, Pa., 1985.
13. PIERCHALA, C. E. An improvement for the McGill University random number package. *Computational Statistics and Data Analysis 2* (1985), 317–322.
14. SAS User's Guide: Basics, Version 5, SAS Institute Inc., Cary, N.C., 1985.
15. Turbo-Pascal Version 3.0 Reference Manual, Borland International, Scotts Valley, Calif., 1985.
16. VAX-11 Pascal Language Reference Manual, Version 3.5, Digital Equipment Corporation, 1987.
17. WICHMANN, B. A., AND HILL, I. D. An efficient and portable pseudo-random number generator. *Appl. Stat., 31* (1982), 188–190.

Received April 1987; revised June 1988; accepted January 1990