Simulation Modeling and Statistical Computing

Richard E. Nance Editor

Efficient and Portable Combined Random Number Generators

PIERRE L'ECUYER

ABSTRACT: In this paper we present an efficient way to combine two or more Multiplicative Linear Congruential Generators (MLCGs) and propose several new generators. The individual MLCGs, making up the proposed combined generators, satisfy stringent theoretical criteria for the quality of the sequence they produce (based on the Spectral Test) and are easy to implement in a portable way. The proposed simple combination method is new and produces a generator whose period is the least common multiple of the individual periods. Each proposed generator has been submitted to a comprehensive battery of statistical tests. We also describe portable implementations, using 16-bit or 32-bit integer arithmetic. The proposed generators have most of the beneficial properties of MLCGs. For example, each generator can be split into many independent generators and it is easy to skip a long subsequence of numbers without doing the work of generating them all.

1. INTRODUCTION

Random number generators are used in many areas including computer simulation, Monte-Carlo techniques in numerical analysis, test problem generation for the performance evaluation of computer algorithms, statistical sampling, and so on. Despite the large amount of theoretical research already done on this subject, many of the generators currently in use, especially those on the microcomputers, are seriously flawed [15]. Even some recently proposed [3, 20] or evaluated [6, 7] generators have a very weak theoretical justification. The aim of this paper is to propose an efficient way to combine two or more *random number generators* to obtain a new, hopefully better one.

All practical "random number" generators on computers are actually simple deterministic computer programs producing a periodic sequence of numbers that should look "apparently random." A generator is defined by a finite *state space* S, a function $f: S \longrightarrow S$ and an initial state s_0 called the *seed*. The state of the generator evolves according to the recursion

$$s_i := f(s_{i-1}), \qquad i = 1, 2, 3, \ldots \tag{1}$$

and the *current state* s_i at stage i is usually transformed into a real value between 0 and 1, according to

$$U_i := g(s_i) \tag{2}$$

where $g: S \longrightarrow (0, 1)$. The *period* of the generator is the smallest positive integer p such that

$$s_{i+p} = s_i \qquad \text{for all } i > \nu \tag{3}$$

for some integer $\nu \geq 0$.

It is well accepted [2, 11] that to obtain a good generator, the choice of f and g should be based on a firm theoretical ground, and before being used for practical applications, the generator should be submitted to a comprehensive set of statistical tests. A good implementation of the generator should be reasonably fast, portable, and use few computer memory words [2, 19].

The most commonly employed generator today is the Lehmer *linear congruential generator* (LCG), for which

$$f(s) = (as + c) \text{ MOD } m; \qquad g(s) = s/m; \tag{4}$$

where the *modulus* m and the *multiplier* $a < m$ are positive integers; and the constant $c < m$ is a nonnegative integer. One usually chooses $c = 0$, in which case the generator is called *multiplicative linear congruential generator* (MLCG) and its *state space* is $S = \{1, 2, \ldots, m - 1\}$.

A MLCG has maximal period ($p = m - 1$) if m is prime and a is a primitive element modulo m [11, p. 19].

Other generators based on linear recursion modulo 2 have been proposed. Tausworthe generators [2, 11] are rather slow in their original form [10]. The generalized feedback shift register generators [10] are faster and have been shown to enjoy good global properties when the parameters are well chosen and when they are based on a primitive polynomial of very large degree. One shortcoming of these generators is that since the state s is a large array, they use a rather large amount of computer memory.

Many disjoint random number subsequences are often required in simulation studies; for example, to facilitate synchronization for variance reduction, or to make independent replications [2]. Such independent subsequences can be produced efficiently by "splitting" a single underlying generator, provided that seeds can be chosen regularly spaced and far enough apart in the cycle to insure that the sequences 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). That can be done quite easily for a MLCG, since

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

For any given j, (a^j MOD m) can be precomputed and (5) can be implemented like any regular MLCG.

On smaller word size machines, MLCGs with large modulus are tricky to implement, while MLCGs with smaller modulus (e.g. m smaller than the largest integer representable on the machine) have periods that are too short to be used safely for serious applications. Various methods have been proposed for combining two or more pseudo-random number generators [2, 16, 21]. The only *mathematically demonstrated* improvement of the combined generators over their components is a much longer period. Beyond this, the combination is an intuitively appealing heuristic supported by both empirical tests and the fact that certain demonstrable pathologies in the components are not apparent in the hybrids (see Figure 4). Some of the combination methods, like bitwise addition modulo 2 [2], apply to LCGs having the same modulus; but if the individual LCGs have full periods, both they and the combined generator have the same period, so, the period is not increased. Other methods, like shuffling [2, 16], do increase the period, but produce a generator that does not seem to have an efficient way to skip a long subsequence of values.

In Section 2, we propose a simple way to combine two or more MLCGs to obtain a generator whose period is the least common multiple of the individual periods. Skipping a fixed number of values can be done quite easily with the combined generator; it suffices to do it with each individual MLCG.

Efficient and portable implementations of MLCGs are not always easy to program in a high level language. Wichmann and Hill [21] and Bratley et al. [2] have proposed a very efficient way to implement a portable MLCG with modulus m using only $\lceil \log_2(m + 1) \rceil$-bit integer arithmetic, when a satisfies:

$$a^2 < m. \qquad (6)$$

In Section 3, we describe efficient ways to code portable implementations for our combined generators by making use of the ideas introduced in [2, 21].

Marsaglia [13] pointed out a theoretical weakness of all LCGs (eq. (4)). He showed that for any given k, all k-tuples $(U_{i+1}, \ldots, U_{i+k})$ of successive values generated by the LCG lie on a set of, at most, $(k!\ m)^{1/k}$ equidistant parallel hyperplanes in the k-dimensional hypercube $(0, 1)^k$. When the number of hyperplanes is too small, obviously, this is a strong limitation to the k-dimensional uniformity. It has become common practice to evaluate LCGs in terms of their induced hyperplanes structures since then.

Fishman and Moore [9] made an exhaustive search of all multipliers for a MLCG with modulus $m = 2^{31} - 1$, to find those for which the maximal distance $d_k(a, m)$ between adjacent parallel hyperplanes in dimension k, for $k = 2, \ldots, 6$, does not exceed the theoretical lower bound on that distance by more than 25 percent.

Unfortunately, none of the multipliers found by Fishman and Moore [9] satisfy the inequality (6). In Section 4 of this paper, we present the results of an extensive search for the best multipliers a in terms of $d_k(a, m)$, for $k = 2, \ldots, 6$, among those that satisfy (6). The search has been made for a collection of prime values of m. We propose multipliers for which $d_k(a, m)$ is satisfactory for every k between 2 and 6. These multipliers are almost as good as those found in [9] and yield generators that are much easier to implement. Our presentation is preceded by a brief review of the principal theoretical considerations that the choice of a multiplier should be based on.

Two new generators, produced by combining some MLCGs retained in Section 4, are proposed in Section 5, one for 32-bit computers and the other for 16-bit computers. Both have been submitted to a battery of statistical tests and their empirical behavior is highly satisfactory.

2. COMBINING LINEAR CONGRUENTIAL GENERATORS

An efficient way to combine many MLCGs to obtain a hopefully better generator is proposed in this section. The method is based on the two following lemmas. Lemma 1 generalizes an informal remark made by Wichmann and Hill [21].

Lemma 1. *Let $W_1, \ldots, W_l$ be l independent discrete random variables such that W_1 is uniform between 0 and $d - 1$, where d is a positive integer*:

$$\Pr(W_1 = n) = \frac{1}{d}. \qquad (7)$$

Then

$$W = \left(\sum_{j=1}^{l} W_j\right) \text{MOD } d \tag{8}$$

follows a discrete uniform probability law between 0 *and* $d - 1$.

Proof. We first show the result for $l = 2$. Let W_2 be discrete (not necessarily uniform) between a and b. For $0 \le n \le d - 1$, we have:

$$\begin{aligned}\Pr(W = n) &= \sum_{k=0}^{\infty} \Pr(W_1 + W_2 = n + kd)\\ &= \sum_{i=a}^{b} \Pr(W_2 = i)\Pr(W_1 = (n - i)\text{MOD } d)\\ &= \frac{1}{d}\sum_{i=a}^{b} \Pr(W_2 = i) = \frac{1}{d}.\end{aligned}$$

In words, whatever the value i taken by W_2, $W = (W_1 + W_2)\text{MOD } d = (W_1 + i)\text{MOD } d$ is uniform between 0 and $d - 1$.

For $l > 2$, define the random variables

$$\begin{aligned}V_2 &= (W_1 + W_2)\text{MOD } d\\ V_3 &= (V_2 + W_3)\text{MOD } d\\ &\vdots\\ V_l &= (V_{l-1} + W_l)\text{MOD } d.\end{aligned}$$

The result for $l = 2$ implies that all those V_j are uniform discrete between 0 and $d - 1$ and since $W = V_l$, this completes the proof. □

Lemma 2. *Consider a family of* l *generators where for* $j = 1, \ldots, l$, *generator* j *has period* p_j *and evolves according to*

$$s_{j,i} := f_j(s_{j,i-1}). \tag{9}$$

Then the period p *of the sequence* $\{s_i = (s_{1,i}, \ldots, s_{l,i}), i = 0, 1, 2, \ldots\}$, *where* $s_0 = (s_{1,0}, \ldots, s_{l,0})$ *is a given seed, is the least common multiple of* $p_1, \ldots, p_l$.

Proof. Each individual generator j has period p_j, so p is a multiple of p_j for each j. If some integer n is a multiple of every p_j, then clearly $s_{i+n} = s_i$ for any $i \ge 0$, which implies that $p \le n$. □

Now we consider the case where each individual generator j is a maximal-period MLCG with modulus m_j and multiplier a_j:

$$s_{j,i} = f_j(s_{j,i-1}) = a_j s_{j,i-1} \text{ MOD } m_j. \tag{10}$$

We combine these generators as suggested by equation (8) with $d = m_1 - 1$. Generator j has period $m_j - 1$ where m_j is prime. Therefore, every $p_j = m_j - 1$ must be even and so an upper bound for p is given by:

$$p \le \frac{\prod_{j=1}^{l}(m_j - 1)}{2^{l-1}} \tag{11}$$

and this bound is attained only when all values of $(m_j - 1)/2$ are relatively prime. During a full cycle, generator 1 takes each value $1, \ldots, m_1 - 1$ only once. Thus, provided that generator 1 is good enough, $s_{1,i} - 1$ may be considered as a uniform discrete variate between 0 and $m_1 - 2$, in the sense of eq. (7). In the rest of this section, we suppose that $s_{1,i}, \ldots, s_{l,i}$ are independent random variables with $s_{1,i}$ uniform on $\{1, \ldots, m_1 - 1\}$. According to Lemma 1,

$$Z_i = \left(\sum_{j=1}^{l} (-1)^{j-1} s_{j,i}\right)\text{MOD}(m_1 - 1) \tag{12}$$

is as a uniform discrete random variate between 0 and $m_1 - 2$ and

$$U_i = \begin{cases} Z_i/m_1 & \text{if } Z_i > 0\\ (m_1 - 1)/m_1 & \text{if } Z_i = 0\end{cases} \tag{13}$$

is therefore a good approximation of a continuous uniform (0, 1) random variable for m_1 large enough.

Only Z_1 needs to be uniform for Lemma 1 to hold. But in practice, Z_1 is not exactly uniform. Therefore, it is (heuristically) more appealing to have all the Z_i as uniform as possible. We keep that before us in the selection of individual MLCGs in Section 5.

3. PORTABLE IMPLEMENTATIONS

Portable generators, implementable in a high level language and producing the same results on any machine with sufficient word length, are highly desirable in most practical situations [2, 19, 21]. Bratley et al. [2, 6.5.2], inspired by Wichmann and Hill [21], propose an efficient way to implement a portable MLCG with modulus m and multiplier a using only integers from $-m$ to $+m$, when $a^2 < m$ (i.e., using only b-bit integer arithmetic if $m < 2^{b-1}$). We present that technique and explain how to use it to implement our combined generators.

Consider a MLCG defined by

$$s_i := f(s_{i-1}) = a s_{i-1}\text{MOD } m \tag{14}$$

where $a^2 < m$. Define

$$q := \lfloor m/a \rfloor \tag{15}$$

$$r := m \text{ MOD } a \tag{16}$$

so that m is decomposed as $m = aq + r$ where $r < a$. For $0 < s < m$, one has

$$\begin{aligned}as \text{ MOD } m &= (as - \lfloor s/q \rfloor m)\text{MOD } m\\ &= (as - \lfloor s/q \rfloor(aq + r))\text{MOD } m\\ &= (a(s - \lfloor s/q \rfloor q) - \lfloor s/q \rfloor r)\text{MOD } m\\ &= (a(s \text{ MOD } q) - \lfloor s/q \rfloor r)\text{MOD } m.\end{aligned} \tag{17}$$

When computing (14) using (17), every intermediate value (integer) during the computation will remain between $-m$ and $+m$. More specifically, $a(s \text{ MOD } q) < aq \le m$, $\lfloor s/q \rfloor r < \lfloor (aq + r)/q \rfloor r \le ar \le a^2 < m$ and both terms are nonnegative, so their difference stays strictly between $-m$ and $+m$.

Assuming that m, a, q and r are global integer constants and that s is a global integer variable holding the current variate, (14) can be implemented as follows (using Pascal-like syntax):

```
k := s DIV q;
s := a * (s − k * q) − k * r;
IF s < 0 THEN s := s + m
```

To obtain a value U between 0 and 1, add the statement

```
U := s * h
```

where h is a precomputed constant equal to $1/m$. This requires only one integer division and four multiplications, which is much more efficient than most other previously proposed implementations (see [12, 14, 17, 18]).

The above technique can be used for each individual MLCG in the implementing of a combined generator as proposed in Section 2, when each of the individual generators satisfies $a^2 < m$. However, additional care must be taken to avoid overflow while computing the sum in eq. (12). The sequence of Pascal statements in Figure 1 implements equations (10, 12–13) using only integers between $-m$ and m. The constant L represents the number l of MLCGs; m[j], a[j], q[j], r[j] and s[j] are the constants and the current variate associated with the MLCG number j, and U denotes the Uniform (0, 1) value that comes out (SNGL converts from INTEGER to REAL).

```
VAR
   U              : REAL;
   j, k, Z        : INTEGER;
   m, a, q, r, s  : ARRAY [1..L] OF INTEGER;
   ...

BEGIN
Z := 0;
FOR j := 1 TO L DO
   BEGIN
   k     := s[j] DIV q[j];
   s[j] := a[j] * (s[j] - k * q[j]) - k * r[j];
   IF s[j] < 0 THEN s[j] := s[j] + m[j];
   IF ODD (j) THEN
      Z := (Z - m[1] + 1) + s[j]
   ELSE
      Z := Z - s[j];
   IF Z < 1 THEN Z := Z + m[1] - 1
   END;
U := SNGL (Z) / SNGL (m[1])
END
```

FIGURE 1. General Code for a Portable Combined Generator

Obviously, the "general" code is not very efficient, it could be optimized for each particular implementation. We use two examples in the next section. On machines where double precision is available, a simplified coding scheme can be used, as shown in Figure 2. It assumes that the arrays m, a and s and the variable Z are stored in floating point with at least $\lceil \log_2(\sum_{j=1}^{l} a_j(m_j - 1)) \rceil$ bits of precision for the mantissa. The function DMOD(x, y) is assumed to yield x MOD y and DBLE converts to double precision. This coding scheme might be slightly faster on some computers having a floating point accelerator.

```
VAR
   U, Z      : DOUBLE
   j         : INTEGER;
   m, a, s   : ARRAY [1..L] OF DOUBLE;
   ...

BEGIN
Z := 0.0;
FOR j := 1 TO L DO
   BEGIN
   s[j] := DBLE (DMOD (a[j] * s[j], m[j]));
   IF ODD (j) THEN Z := Z + s[j] ELSE Z := Z - s[j];
   END;
WHILE Z < 0.5            DO  Z := Z + (m[1] - 1);
WHILE Z > (m[1] - 0.5) DO  Z := Z - (m[1] - 1);
U := Z / m[1]
END
```

FIGURE 2. A More Direct Coding Scheme Using Double Precision

4. SEARCHING FOR GOOD MULTIPLIERS

Multidimensional uniformity of the k-tuples of successive numbers is the ultimate measure of goodness of pseudo-random number generators. In practice, obtaining k-dimensional uniformity for every positive integer k is impossible, but good uniformity for small values of k is a must. Many theoretical tests to measure the uniformity for a given LCG and a given k have been suggested; most of these tests are based on the fact that all k-tuples $P_{i,k} = (U_{i+1}, \ldots, U_{i+k})$ are arranged on parallel hyperplanes and form a lattice structure in the k-dimensional unit hypercube.

The *Spectral Test* [2, 4, 5, 11] determines the maximal distance $d_k(m, a)$ between adjacent parallel hyperplanes, the maximum being taken over all families of parallel hyperplanes that cover all the points $P_{i,k}$. The smaller that maximal distance, the better the generator is, since this implies smaller empty "slices" in the hypercube. However, there is a theoretical lower bound $d_k^*(m)$ on $d_k(m, a)$, that is given here for $2 \le k \le 8$ (see [11, p. 105]):

$$d_k(m, a) \ge d_k^*(m) \overset{def}{=} \begin{cases} (4/3)^{-1/4} m^{-1/2}, & \text{if } k = 2; \\ 2^{-1/6} m^{-1/3}, & \text{if } k = 3; \\ 2^{-1/4} m^{-1/4}, & \text{if } k = 4; \\ 2^{-3/10} m^{-1/5}, & \text{if } k = 5; \\ (64/3)^{-1/12} m^{-1/6}, & \text{if } k = 6; \\ 2^{-3/7} m^{-1/7}, & \text{if } k = 7; \\ 2^{-1/2} m^{-1/8}, & \text{if } k = 8. \end{cases} \tag{18}$$

Normalizing $d_k(m, a)$ as in [9], we obtain the figure of merit

$$S_k(m, a) = \frac{d_k^*(m)}{d_k(m, a)} \tag{19}$$

that lies between 0 and 1. An algorithm to compute $d_k(m, a)$ was proposed by Coveyou and MacPherson [4] and later improved by Knuth [11] and Dieter [5]. The algorithm is described in [11, pp. 98–100].

A second measure of quality, suggested by Marsaglia [13], is the minimal *number of parallel hyperplanes* covering all the points $P_{i,k}$. This number should be high, otherwise large regions will be devoid of points. However, as Knuth [11, p. 92] points out, this measure, although strongly related to $d_k(m, a)$, is "biased by how nearly the slope of the hyperplanes matches the coordinate axes," and is then a less significant criterion.

Niederreiter [17] has shown how to analyse LCGs in terms of their *discrepancy*. The discrepancy in k dimensions can be defined [11, 17] as the maximum absolute difference between the expected number and the observed number of points $P_{i,k}$ in a k-dimensional hyperrectangular box aligned with the axes, the maximum being taken over all such boxes. The discrepancy is defined for any subsequence of length $N \le p$, where p is the period. It can be shown [11] that any LCG that performs well in the Spectral Test will have rather small discrepancy for large enough N (Knuth suggests $N > \sqrt{m}(\log m)^{t+1}$). The measure of discrepancy detects the worst cases only with regard to boxes that are aligned with the axes; it can change significantly when the points are rotated, unlike the result of the (rotationally invariant) Spectral Test. This suggests that the latter test should be considered as more meaningful. No algorithm exists to compute the discrepancy; only expressions giving lower and upper bounds are known and even these bounds cannot be computed in general.

Other measures of goodness have been suggested, such as the *lattice test*, the minimal distance between any two points, measures of packing the lattice with spheres, bounds on the serial correlation, and so on (see [1, 9]). However, as suggested in [9, 11], the results of these tests are either strongly correlated or dominated by those of the Spectral Test, which appears to be the most powerful test known for LCGs. Borosh and Niederreiter [1] have found "optimal" multipliers for moduli 2^n, $6 \le n \le 35$. The "optimality" criterion is simply a measure of uniformity of the empirical distribution of the pairs (U_{i+1}, U_{i+2}) over the unit square.

We applied the Spectral Test for some values of m to find, among all multipliers $a \le \sqrt{m}$ that are primitive elements modulo m, those that perform well in every dimension k between 2 and 6. More specifically, we found those a for which the worst case measure

$$M_6(m, a) \overset{def}{=} \min_{2 \le k \le 6} S_k(m, a) \tag{20}$$

is the largest (the closest to unity). A summary of our results is given in Table I. The first and seventh values of m (2147483647 and 32749) are the largest primes smaller than 2^{31} and 2^{15} respectively (that can be represented in two's-complement 32-bit and 16-bit integer arithmetic respectively). The second to sixth values are those for which $\sup_{a \le \sqrt{m}} M_6(m, a)$ is the largest among the 50 largest primes smaller than 2^{31} and the last five are those for which $\sup_{a \le \sqrt{m}} M_6(m, a)$ is the largest among the 100 largest primes smaller than 2^{15}.

For each m, the first multiplier a in Table I is the best according to $M_6(m, a)$, among those for which $a \le \sqrt{m}$.

For $m = 2147483647$, the multipliers $a = 742938285$ and $a = 950706376$ are the two best overall according to [9]; $a = 16807$ has been suggested by Lewis et al. [12] and is also recommended in [2, 19]; $a = 630360016$ is used in the SIMSCRIPT II.5 language and is recommended in [14]. For $m = 32749$, $a = 219$ is the best multiplier overall, but is larger than $\sqrt{m}$. However, the last five combinations of m and a given in Table I, have a better figure of merit. In general, the constraint $a \le \sqrt{m}$ is not costly in terms of the best achievable value of $M_6(m, a)$.

For a simple and easily implementable MLCG on a 32-bit computer, we suggest $m = 2147483399$ and $a = 40692$. On a 16-bit computer, we hesitate to recommend the general use of any simple MLCG with $m < 2^{15}$, since the lattice structure is too coarse and the period is too short. Combined generators offer much longer periods and we advocate them for both 16-bit and 32-bit computers.

TABLE I. Computed Values of $S_k(m, a)$ and $M_6(m, a)$ for the Spectral Test.

m	a	$S_2(m, a)$	$S_3(m, a)$	$S_4(m, a)$	$S_5(m, a)$	$S_6(m, a)$	$M_6(m, a)$
2147483647	39373	.7907	.7549	.7866	.7580	.7545	.7545
2147483647	742938285	.8673	.8607	.8627	.8319	.8341	.8319
2147483647	950706376	.8574	.8985	.8692	.8337	.8274	.8274
2147483647	16807	.3375	.4412	.5752	.7361	.6454	.3375
2147483647	630360016	.8212	.4317	.7833	.8021	.5700	.4317
2147483563	40014	.8035	.8357	.7885	.8281	.8081	.7885
2147483399	40692	.8172	.8180	.8051	.8912	.8181	.8051
2147482811	41546	.8343	.7870	.8112	.8085	.8206	.7870
2147482801	42024	.8439	.8111	.8568	.7830	.8101	.7830
2147482739	45742	.9186	.8512	.7833	.8201	.7991	.7833
32749	162	.8331	.7959	.7100	.6581	.7628	.6581
32749	219	.9299	.7930	.7263	.7180	.7628	.7180
32363	157	.8122	.8507	.8270	.7818	.7885	.7818
32143	160	.8305	.7545	.8067	.7279	.7774	.7279
32119	172	.8931	.7195	.7352	.7763	.7401	.7195
31727	146	.7628	.7219	.7266	.7579	.7288	.7219
31657	142	.7427	.7625	.8244	.7853	.7794	.7427

TABLE II. Theoretical Lower Bounds on $d_k(m, a)$

	m	$d_2^*(m)$	$d_3^*(m)$	$d_4^*(m)$	$d_5^*(m)$	$d_6^*(m)$
2^{31} =	2147483648	.0000201	.000690	.00391	.01105	.02157
	2147482739	.0000201	.000690	.00391	.01105	.02157
2^{15} =	32768	.0051409	.027840	.06251	.10154	.13700
	31657	.0052303	.028163	.06304	.10223	.13777

The values of the theoretical lower bounds on $d_k(m, a)$ given by eq. (18) appear in Table II for four values of m. These values give a better insight about the significance of the results of Table I. Notice that for close values of m, these bounds are about the same. For example, for $m = 2^{31} = 2147483648$ and $m = 2147482739$, the figures shown in Table II are exactly the same. This means that for all values of m between these two, the numbers $S_k(m, a)$ and $M_6(m, a)$ have about the same significance. For values of m near 2^{15}, the difference is more perceptible, but still relatively small.

5. TWO NEW GENERATORS

From the results of the previous sections, we can now propose two new combined generators. For a b-bit word length, we want $m_j < 2^{b-1}$ and $a_j \leq \sqrt{m_j}$ for every j, so that every individual generator can be easily implemented in a portable and efficient way using the technique proposed in [2, 21].

For 32-bit computers, we suggest $l = 2$, $m_1 = 2147483563$, $a_1 = 40014$, $m_2 = 2147483399$ and $a_2 = 40692$. These two individual MLCGs are excellent according to the Spectral Test (see Table I). Furthermore, $(m_1 - 1)/2 = 3 \times 7 \times 631 \times 81031$ and $(m_2 - 1)/2 = 19 \times 31 \times 1019 \times 1789$ are relatively prime and the combined generator has period $p = (m_1 - 1)(m_2 - 1)/2 \approx 2.30584 \times 10^{18}$.

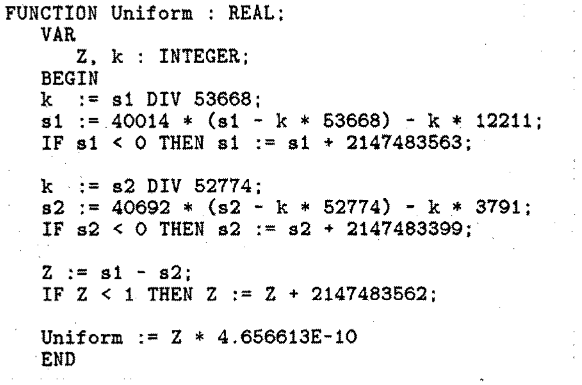
For 16-bit computers, we suggest $l = 3$ and pick the three MLCGs defined by $m_1 = 32363$, $a_1 = 157$, $m_2 = 31727$, $a_2 = 146$, $m_3 = 31657$ and $a_3 = 142$. They all perform very well in the Spectral Test and the values of $(m_1 - 1)/2 = 11 \times 1471$, $(m_2 - 1)/2 = 29 \times 547$ and $(m_3 - 1)/2 = 2 \times 2 \times 3 \times 1319$ are relatively prime. The period of the combined generator is then $p = (m_1 - 1)(m_2 - 1)(m_3 - 1)/4 \approx 8.12544 \times 10^{12}$. The generator with $m = 32143$ and $a = 160$ is not selected despite its good performance in the Spectral Test since 32142/2 has 11 as a common factor with $(m_1 - 1)/2$.

The values of q and r for each individual MLCG are given in Table III.

TABLE III. The Values of m, a, q and r for the Five Retained MLCGs

m	a	q	r
2147483563	40014	53668	12211
2147483399	40692	52774	3791
32363	157	206	21
31727	146	217	45
31657	142	222	133

Figure 3 gives a Pascal function implementing the first proposed combined generator, using an optimized version of the code given in Figure 1. It works as long as the machine can represent all integers in the range $[-2^{31} + 85, 2^{31} - 85]$. The integer variables s1 and s2 are global and hold the current variates. Before the first call to Uniform, they must be initialized to values in the range [1, 2147483562] and [1, 2147483398] respectively. Notice that the function will never return 0.0 or 1.0, as long as REAL variables have at least 23-bit mantissa (this is the case for most 32-bit machines). In the second edition of their book, Bratley, Fox and Schrage [2] adopt this generator. They provide a FORTRAN implementation and specific seeds to generate disjoint streams. Such specific seeds, spaced say 2^d values apart in the sequence, can be computed as follows: for each of the two MLCG components, (i) choose any seed in the proper range; (ii) precompute a^{2^d} MOD m, where a and m are the constants defining this MLCG; and (iii) use eq. (5). An efficient way to precompute a^{2^d} MOD m is to start with a and square it d times modulo m. Notice that this squaring, as well as the product in eq. (5), must be done in extended precision arithmetic ($\geq$ 62 bits).



```
FUNCTION Uniform : REAL;
   VAR
      Z, k : INTEGER;
   BEGIN
   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;

   Z := s1 - s2;
   IF Z < 1 THEN Z := Z + 2147483562;

   Uniform := Z * 4.656613E-10
   END
```

FIGURE 3. A Portable Generator for 32-bit Computers

Figure 4 gives a portable code for the other proposed combined generator for 16-bit computers. It assumes that integers in the range [−32363, 32363] are well represented. The (global) integer variables s1, s2 and s3 must be initialized to values in the range [1, 32362], [1, 31726] and [1, 31656] respectively.

Wichmann and Hill [21] have proposed a different portable combined generator, for 16-bit computers. It generates three Uniform (0, 1) values using three "independent" MLCGs, and takes the sum modulo one. The three individual MLCGs have modulus 30269, 30307 and 30323 and multipliers 171, 172 and 170 respectively. Their $M_6(a, m)$ values are .1830, .6228 and .4639 respectively, which is rather low compared to the MLCGs proposed here. According to Lemma 2, the generator proposed in [21] has period $p = 6.95 \times 10^{12}$ instead of $p > 2.78 \times 10^{13}$ as claimed.

```
FUNCTION Uniform : REAL;
   VAR
      Z, k  :  INTEGER;
   BEGIN
   k  := s1 DIV 206;
   s1 := 157 * (s1 - k * 206) - k * 21;
   IF s1 < 0 THEN s1 := s1 + 32363;

   k  := s2 DIV 217;
   s2 := 146 * (s2 - k * 217) - k * 45;
   IF s2 < 0 THEN s2 := s2 + 31727;

   k  := s3 DIV 222;
   s3 := 142 * (s3 - k * 222) - k * 133;
   IF s3 < 0 THEN s3 := s3 + 31657;

   Z  := s1 - s2;
   IF Z > 706 THEN Z := Z - 32362;
   Z := Z + s3;
   IF Z < 1 THEN Z := Z + 32362;

   Uniform :=  Z * 3.0899E-5
   END
```

FIGURE 4. A Portable Generator for 16-bit Computers

As proposed here, the combinations are performed in integer arithmetic. In [21], the values are transformed into reals between 0 and 1 before being combined. Combination in integer arithmetic is faster, and is also advantageous when one uses directly the integer Z_i instead of the floating point number U_i (for example, when generating uniform random integers over an arbitrary interval; see [2], section 6.7.1). On the other hand, for the 16-bit generator proposed here, the number of possible output values is only 32362 (the number of possible values of Z_i); for the generator proposed in [21], that number depends on the floating point representation and is generally much higher.

6. EMPIRICAL TESTING

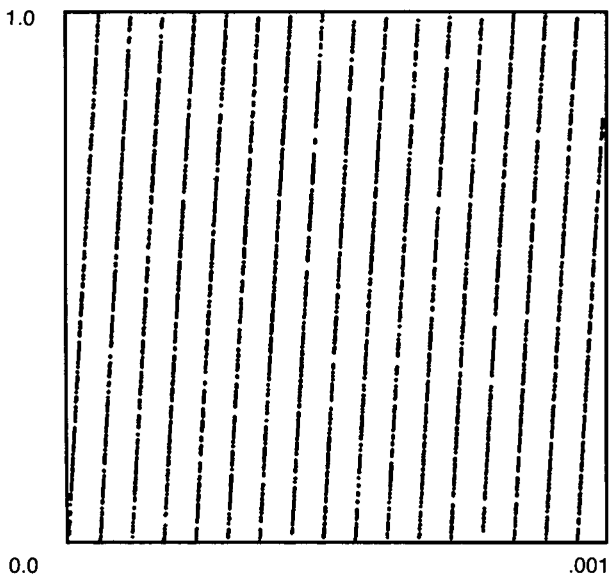
The two proposed combined generators and one simple MLCG have been submitted to a comprehensive battery of statistical tests described in Knuth [11, pp. 59–73]. Each test produces a statistic that, under the null hypothesis H_0 that the generator is good, has a known theoretical probability distribution. Furthermore, every test has been repeated N times and the empirical distribution of the values of the statistics has been compared to the theoretical distribution using the classical Kolmogorov–Smirnov (KS) test. Thus the final result is the value s of a KS statistic S. A generator fails the test if the observed descriptive level $\delta = \Pr(S \le s \mid H_0)$ is "too small."

We performed 21 different tests on the three generators. They are described below using the notation of Knuth [11] for their parameters. Here, n denotes the number of observations during a given run and N denotes the number of runs. These tests involve billions of pseudo-random numbers and took more than 200 hours of CPU time on a VAX-11/780.

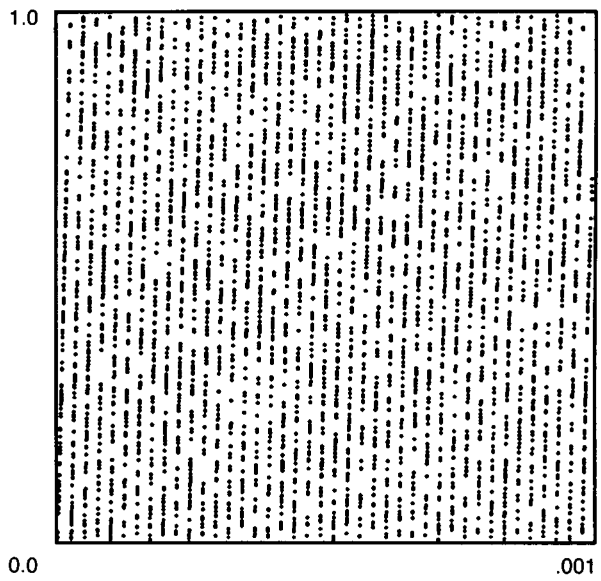
(1) Equidistribution test, using chi-square, $d = 64$, $n = 1000$, $N = 10000$.
(2) Equidistribution test, using chi-square, $d = 256$, $n = 10000$, $N = 10000$.
(3) Serial test with pairs (2-dimensional), $d = 64$, $n = 100000$, $N = 1000$.
(4) Serial test with triplets (3-dimensional), $d = 16$, $n = 100000$, $N = 1000$.
(5) Serial test with quadruplets (4-dimensional), $d = 8$, $n = 100000$, $N = 1000$.
(6) Gap test, $\alpha = 0$, $\beta = .05$, $t = 15$, $n = 10000$, $N = 1000$.
(7) Gap test, $\alpha = .95$, $\beta = 1$, $t = 15$, $n = 10000$, $N = 1000$.
(8) Gap test, $\alpha = 1/3$, $\beta = 2/3$, $t = 10$, $n = 10000$, $N = 1000$.
(9) Poker test, $k = 4$, $d = 4$, $n = 10000$, $N = 1000$.
(10) Poker test, $k = 6$, $d = 4$, $n = 10000$, $N = 1000$.
(11) Poker test, $k = 6$, $d = 8$, $n = 10000$, $N = 1000$.
(12) Poker test, $k = 8$, $d = 16$, $n = 10000$, $N = 1000$.
(13) Coupon's collector test, $d = 5$, $t = 25$, $n = 10000$, $N = 1000$.
(14) Coupon's collector test, $d = 10$, $t = 40$, $n = 10000$, $N = 1000$.
(15) Permutation test, $t = 3$, $n = 10000$, $N = 1000$.
(16) Permutation test, $t = 5$, $n = 10000$, $N = 1000$.
(17) Runs-up test, $n = 100000$, $N = 1000$.
(18) Maximum-of-t test, $t = 8$, $d = 128$, $n = 10000$, $N = 1000$.
(19) Collision test, 6 dimensions, $d = 8$, $n = 20000$, $N = 100$.
(20) Collision test, 10 dimensions, $d = 4$, $n = 20000$, $N = 100$.
(21) Collision test, 20 dimensions, $d = 2$, $n = 20000$, $N = 100$.

The results of the tests appear in Table IV, where δ_1 and δ_2 represent the observed value of δ for the first (for 32-bit) and second (for 16-bit) proposed combined generators respectively. δ_3 is the observed value of δ for the simple MLCG with modulus $m = 2147483399$ and multiplier $a = 40692$ suggested in Section 4.

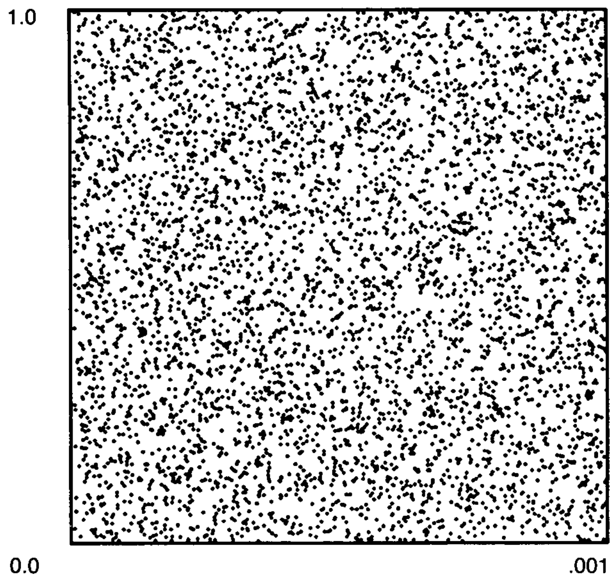
The initial seeds for each test were $s1 = 12345$ and $s2 = 67890$ for the first generator; $s1 = 12$, $s2 = 23$ and $s3 = 34$ for the second generator; and $s1 = 12345$ for the third generator. We arbitrarily declare as "suspects" the values of δ smaller than 0.05. These low values, marked with a $*$ superscript in Table IV, could have been produced by chance (it should occur 5 percent of the time under H_0) or may indicate flaws in the generators. To probe further, we repeated those tests that produced the "suspect" values, using different (disjoint) random number streams. To guarantee that the streams were disjoint, we used as starting seeds the final values at the end of the corresponding "first trial" tests. The



(a) The MLCG with $m = 2^{31} - 1$ and $a = 16807$.



(b) The MLCG with $m = 2147483399$ and $a = 40692$.



(c) The proposed combined generator.

FIGURE 5. A Thin Slice of the Plot of Output Pairs for Three 32-bit Generators

results of these "second trials" appear in Table V.

The combined 32-bit generator produced no suspect value. For the combined 16-bit generator, the new value of δ is no more suspect. The result of test 13 (coupon's collector) for the 32-bit MLCG is still suspect. The MLCG also had 3 suspect values on the first trials, compared to 0 and 1 respectively for the combined generators. In summary, we conclude that the empirical tests support the latter.

Figure 5 gives a partial graphical illustration of the two-dimensional behavior of three generators. For each of the three plots, five million pairs of consecutive numbers were generated and placed in the unit square. A thin slice of the surface of the square, .001 wide by 1.0 high, was then cut on its left side and stretched out horizontally. Thus, each part of Figure 4 contains only the pairs (U_{i+1}, U_{i+2}) such that $U_{i+1} < .001$, (i.e., approximately 5000 points).

In Figure 5, Box (a) shows the behavior of the often recommended MLCG with $m = 2^{31} - 1$ and $a = 16807$. The lattice structure is quite clear. It looks a little better in Box (b), which shows the behavior of the MLCG

TABLE IV. Results of the Empirical Tests

Test	δ_1	δ_2	δ_3
1.	.0961	.1920	.0510
2.	.7984	.6096	.0582
3.	.7388	.6461	.0071*
4.	.4399	.6507	.7647
5.	.7530	.6466	.9414
6.	.8818	.1243	.0479*
7.	.0751	.3509	.7794
8.	.1881	.9699	.7275
9.	.1879	.1898	.4054
10.	.6358	.6920	.6765
11.	.3925	.3960	.3645
12.	.3395	.4591	.3491
13.	.9390	.1945	.0255*
14.	.4053	.2914	.4543
15.	.8859	.9357	.2557
16.	.3516	.2503	.0640
17.	.1775	.9895	.0930
18.	.8703	.0252*	.7532
19.	.9341	.4382	.1216
20.	.2101	.8435	.7369
21.	.1019	.2178	.4633

*(smaller than .05)

(continued on p. 774)

61. Shostack, C., and Eddy, C. Management by computer graphics. *Harvard Bus. Rev. 49*, 6 (Nov.-Dec. 1971), 52–63.
62. Simon, H.A. Information processing models of cognition. *Ann. Rev. of Psyc. 30* (1979), 363–396.
63. Strickland, R.G. *A Study of the Possibilities of Graphs as a Means of Instruction in the First Four Grades of Elementary School.* Columbia University, 1938. Vol. 30, 311.
64. Takeuchi, H., and Schmidt A.H. New promise of computer graphics. *Harvard Bus. Rev. 58*, 1 (Jan.-Feb. 1980), 122–131.
65. Thiel, C.T. The big boom in computer graphics. *Infosys. 29*, 5 (1982), 48–56.
66. Tullis, T.S. An evaluation of alphanumeric, graphic, and color information display. *Human Factors 23*, 5 (Oct. 1981), 541–550.
67. Vernon, M.D. Learning from graphical material. *Brit. J. of Psyc. 36*, Part 3 (May 1946), 145–158.
68. Vernon, M.D. The use and value of graphical material in presenting quantitative data. *Occup. Psyc. 26* (1952), 22–34.
69. Vicino, F.L., and Ringel, S. Decision-making with Updated Graphic vs. Alphanumeric Information. Wash., D.C.: Army Pers. Res. Office, Techn. Res. Note 178, Nov. 1966.
70. Vogel, D.R., Dickson, G.W., and Lehman, J.A. "Persuasion and the Role of Visual Presentation Support: The UM/3M Study." MISRC-WP-86-11, June 1986.
71. Wainer, H., and Reiser, M. Assessing the efficacy of visual displays. *Proc. of the Am. Stat. Assoc.*, Soc. Stat. Sect., 1, Part I (Aug. 1976), 89–92.
72. Washburne, J.N. An experimental study of various graphic, tabular and textural methods of presenting quantitative material. *J. of Ed. Psyc. 18*, 6 (Sept. 1927), 361–376.
73. Watson, C.J., and Driver, R.W. The influence of computer graphics on the recall of information. *MIS Qrtly. 7*, 1 (Mar. 1983), 45–53.
74. Wilcox, W. Numbers and the news: Graph, table or text? *Journalism Qrtly. 41*, 1 (Winter 1964), 38–44.
75. Welsch, R.E. Graphics for data analysis. *Comp. and Graphics 2*, 1 (1976), 31–37.

CR Categories and Subject Descriptors: H.3.0 [**Information Storage and Retrieval**]: General; I.3.0 [**Computer Graphics**]: General.

General Terms: Experimentation, Graphics, Management

Additional Key Words and Phrases: Business, decision making, decision support, guidelines, research

Received 2/86; revised 1/87; accepted 11/87

Authors' Addresses: Sirkka L. Jarvenpaa, Department of Management Science and Information Systems, University of Texas at Austin, Austin, TX 78712; Gary W. Dickson, Department of Business Computer Information Systems, College of Business, St. Cloud State University, St. Cloud, MN 56301.

L'Ecuyer *(continued from p. 749)*

TABLE V. Results of the "Second Trial" Tests

Generator	Test no.	Previous δ	New δ
Combined 16-bit	18.	.0252	.4139
32-bit MLCG	3.	.0071	.1098
32-bit MLCG	6.	.0479	.1967
32-bit MLCG	13.	.0255	.0228

with $m = 2147483399$ and $a = 40692$. Box (c) is the output from the proposed 32-bit combined generator: No lattice structure is apparent. These graphics are just more empirical evidence supporting the combination.

Acknowledgments. This work has been supported by NSERC-Canada grant number A5463 and FCAR-Quebec grant number EQ2831. I would like to acknowledge the many helpful suggestions by Professor B. L. Fox, L. E. Schrage, C. Dupuis, and T. Vo-Dai. I would also like to thank Pierre Poulin, Eric Boucher, Brigitte Mercier, Sylvie Prigent and Marc de la Durantaye, who wrote the large amount of Pascal code required for this research.

REFERENCES

Note: Reference [8] is not cited in text.

1. Borosh, S., and Niederreiter, H. Optimal multipliers for pseudo-random number generation by the linear congruential method. *BIT 23* (1983), 65–74.
2. Bratley, P., Fox, B.L., and Schrage, L.E. *A Guide to Simulation.* Springer-Verlag, New York, N.Y., 2nd ed., 1987.
3. Clark, R.N. A Pseudorandom Number Generator. *Simulation 45*, 5 (Nov. 1985), 252–255.
4. Coveyou, R.R., and MacPherson, R.D. Fourier analysis of uniform random number generators. *J. ACM 14* (Jan. 1967), 100–119.
5. Dieter, U. How to calculate shortest vectors in a lattice. *Math. Comput. 29* (July 1975), 827–833.
6. Dudewicz, E.J., Karian, Z.A., and Marshall, R.J., III. Random number generation on microcomputers. *Modeling and Simulation on Microcomputers: 1985*, The Society for Computer Simulation, 1985, pp. 9–14.
7. Figiel, K.D., and Sule, D.R. New lagged product test for random number generators. *Comput. Ind. Eng. 9*, 3 (Mar. 1985), 287–296.
8. Fishman, G.S., and Moore, L.S., III. A statistical evaluation of multiplicative congruential random number generators with modulus 2^{31}–1. *J. Am. Stat. Assoc. 77* (Mar. 1982), 129–136.
9. Fishman, G.S., and Moore, L.S., III. An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$. *SIAM J. Sci. Stat. Comput. 7*, 1 (Jan. 1986), 24–45.
10. Fushimi, M., and Tezuka, S. The *k*-distribution of generalized feedback shift register pseudorandom numbers. *Commun. ACM 26*, 7 (July 1983), 516–523.
11. Knuth, D.E. *The Art of Computer Programming—Seminumerical Algorithms.* vol. 2, 2nd ed., Addison-Wesley, Reading, Mass. 1981.
12. Lewis, P.A.W., Goodman, A.S., and Miller, J.M. A pseudo-random number generator for the system/360. *IBM Syst. J. 8*, 2 (1969), 136–146.
13. Marsaglia, G. Random numbers fall mainly in the planes. *Proc. Nat. Acad. Sci. 61* (Sept. 1968), 25–28.
14. Marse, K. and Roberts, S.D. Implementing a portable FORTRAN uniform (0, 1) generator. *Simulation 41*, 4 (Oct. 1983), 135–139.
15. Modianos, D.T., Scott, R.C., and Cornwell, L.W. Random number generation on microcomputers. *Interfaces 14*, 2 (Mar.–April 1984), 81–87.
16. Nance, R.E., and Overstreet, C., Jr. Some experimental observations on the behavior of composite random number generators. *Oper. Res. 26*, 5 (Sept.–Oct. 1978), 915–935.
17. Niederreiter, H. Quasi-Monte Carlo methods and pseudo-random numbers. *Bull. Amer. Math. Soc. 84*, 6 (Nov. 1978), 957–1041.
18. 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.
19. Schrage, L. A more portable Fortran random number generator. *ACM Trans. Math. Soft. 5*, 2 (June 1979), 132–138.
20. Thesen, A. An efficient generator of uniformly distributed random variates between zero and one. *Simulation 44*, 1 (Jan. 1985), 17–22.
21. Wichmann, B.A and Hill, I.D. An efficient and portable pseudo-random number generator. *Appl. Stat. 31* (1982), 188–190.

CR Categories and Subject Descriptors: G.3 [**Probability and Statistics**]: *Random number generation*

General Terms: Algorithms, performance

Additional Key Words and Phrases: Combined generators, empirical tests, multiplicative linear congruential generators, portability, spectral test

Received 7/86; revised 2/87; accepted 3/87

Authors' Present Address: Pierre L'Ecuyer, Département d'informatique, Université Laval, Ste-Foy, Quebec, Canada, G1K 7P4.