Long version, May 4, 1998

GOOD PARAMETERS AND IMPLEMENTATIONS FOR COMBINED MULTIPLE RECURSIVE RANDOM NUMBER GENERATORS

PIERRE L'ECUYER

Département d'Informatique et de Recherche Opérationnelle
Université de Montréal, C.P. 6128, Succ. Centre-Ville, Montréal, H3C 3J7, Canada

Combining parallel multiple recursive sequences provides an efficient way of implementing random number generators with long periods and good structural properties. Such generators are statistically more robust than simple linear congruential generators that fit into a computer word. We made extensive computer searches for good parameter sets, with respect to the spectral test, for combined multiple recursive generators of different sizes. We also compare different implementations and give a specific code in C that is faster than previous implementations of similar generators.

It is now recognized that random number generators (RNGs) should have huge periods, several orders of magnitude larger than whatever can be used in practice (L'Ecuyer 1994, L'Ecuyer 1998b, Ripley 1987). For example, *all* full-period linear congruential generators (LCGs), or multiple recursive generators (MRGs), fail decisively some statistical tests that use approximately $\sqrt{\rho}$ random numbers, where ρ is the period length (see, e.g., L'Ecuyer, Cordeau, and Simard 1997, L'Ecuyer and Hellekalek 1998 for the LCGs). To be reasonably safe, the period length of a general purpose generator must exceed 2^{100} or so, and preferably more. And a long period is not sufficient. Good structural properties are also needed. If the aim is to imitate a sequence of i.i.d. $U(0,1)$ (independent and identically distributed random variables, uniform over the interval $[0,1]$), the set $T_t = \{\boldsymbol{u}_n = (u_n, \ldots, u_{n+t-1}),\ n \geq 0\}$, of all vectors of t successive output values over all the generator's cycles, should be uniformly distributed over the t-dimensional unit hypercube $[0,1]^t$, for all t (ideally). If the seed is random, this set T_t can be viewed as a *sample space* from which some points are drawn. In practice, the structural properties of T_t can be analyzed via the spectral test, for t up to 30 or so.

A *multiple recursive generator* (MRG) of *order* k is defined by the linear recurrence:

$$\begin{aligned} x_n &= (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod m; \\ u_n &= x_n/m, \end{aligned} \tag{1}$$

where m and k are positive integers, and each a_i belongs to $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$ (see Grube 1973, Niederreiter 1992). The recurrence (1) has maximal period length $m^k - 1$, attained if and only if m is prime and the characteristic polynomial $P(z) = z^k - a_1 z^{k-1} - \cdots - a_k$ is primitive (i.e., the powers of z, modulo $P(z)$ and m, run through all nonzero polynomials of degree less than k with coefficients in $\mathbb{Z}^m$). The latter can be achieved most economically with only two nonzero coefficients, say a_r and a_k with $1 \leq r < k$. The recurrence is generally easier to implement when these coefficients are small. However, a *necessary* condition for a good figure of merit with respect to the spectral test is that $\sum_{i=1}^{k} a_i^2$ be large (Grube 1973, L'Ecuyer 1997). To reconcile these conflicting requirements, L'Ecuyer (1996) proposed combined MRGs (CMRGs), where the components are carefully selected so that the combined generator has good structural properties, while each component remains easy to implement in an efficient manner. Such a CMRG turns out to be equivalent (or approximately equivalent, depending on the type of combination) to an MRG with a large composite modulus, equal to the product of the moduli of its components. The recurrence of the CMRG can have many large coefficients even if the components have only two small nonzero coefficients. L'Ecuyer (1996) gave a few examples of CMRGs, but only one of these (Example 4) was a recommendable generator, with two components of order 3, period length approximately 2^{185}, and with the parameters chosen specifically for an implementation using 31-bit integer arithmetic with the "approximate factoring" method. That generator behaves well with respect to the spectral test in up to 20 dimensions.

The aim of this paper is to provide good CMRGs of different sizes, selected via the spectral test up to 32 (or 24) dimensions, and a faster implementation than in L'Ecuyer (1996) using floating-point arithmetic. Why do we need different parameter sets? Firstly, different types of implementations require different constraints on the modulus and multipliers. For example, a floating-point implementation with 53 bits of precision allows moduli of more than 31 bits and this can be exploited to increase the period length for free. Secondly, as 64-bit computers get more widespread, there is demand for generators implemented in 64-bit integer arithmetic. Tables of good parameters for such generators must be made available. Thirdly, RNGs are somewhat like cars: a single model and single size for the entire world is not the most satisfactory solution. Some people want a fast and relatively small RNG, while others prefer a bigger and more robust one, with longer period and good equidistribution properties in larger dimensions. Naively, one could think that an RNG with period length near 2^{185} is big enough for any conceivable application. But note that 185 (selected) bits of the RNG's sequence are enough to determine all the others, so this sequence has a lot of structure, and for this reason some might want a bigger number than 185.

The tables provided here are the partial results of an extensive computer search that took more than a year of CPU time on *SUN Sparcstations* using the software described in L'Ecuyer and Couture (1997). The next section recalls some notation, defines the figures of merit that we use, and explains our search strategies. Section 2 reports the results. Section 3 provides an implementation in C and gives timing comparisons. The C code is also available at `ftp.iro.umontreal.ca` in directory `pub/simulation/lecuyer/combmrg2`. Look for the file `combmrg2.c`. A shorter version of this paper will appear as L'Ecuyer (1998a).

1. Notation, Selection Criteria, and Implementation Conditions

The RNGs considered in this paper combine J copies of (1), that is:

$$x_{j,n} = (a_{j,1}x_{j,n-1} + \cdots + a_{j,k}x_{j,n-k}) \bmod m_j \qquad (2)$$

for $j = 1, \ldots, J$, where the m_j are distinct primes and the jth recurrence has order k and period length $m_j^k - 1$. Let $\delta_1, \ldots, \delta_J$ be arbitrary integers such that δ_j is relatively prime to m_j for each j, and define:

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

$$z_n = \left(\sum_{j=1}^{J} \delta_j x_{j,n}\right) \bmod m_1, \qquad (4)$$

$$\tilde{u}_n = z_n/m_1. \qquad (5)$$

The sequences $\{w_n,\, n \geq 0\}$ and $\{\tilde{u}_n,\, n \geq 0\}$ define two different CMRGs which have been studied by L'Ecuyer (1996). In summary, the CMRG (2)–(3) is exactly equivalent to an MRG as in (1) with modulus $m = m_1 \cdots m_J$, and the set T_t mentioned in the introduction is the intersection of a lattice with the unit hypercube. The points of T_t lie in successive parallel hyperplanes at a distance d_t of each other. The other CMRG, defined by (4)–(5), is also approximately the same as the first one. In other words, these CMRGs are basically just *special implementations* of an MRG and they can be analyzed by applying the spectral test to this MRG.

We use the figure of merit $M_T = \min_{2 \leq t \leq T} S_t$ for some integer T, where $S_t = (\rho_t m^{k/t} d_t)^{-1}$ and ρ_t is defined as follows. For $t \leq 8$, ρ_t is the γ_t defined in Knuth (1981), page 105, while for $t > 8$, $\rho_t = \exp(R(t)/t)$ where $R(t)$ is Rogers' bound on the density of sphere packings (see Conway and Sloane 1988, page 88, and L'Ecuyer 1998c). S_t and M_T are always between 0 and 1 and we seek generators with M_T close to 1. An S_t close to 0 means that all the points of T_t lie in equidistant parallel hyperplanes that are far apart, leaving thick slices of empty space in between. An M_T close to 1 means that T_t is evenly distributed over the unit hypercube, for all $t \leq T$.

For $J = 2, 3$, $k = 3, 5, 7$, and prime moduli slightly smaller than 2^e for $e = 31$, 32, 63, 64, 127, and 128, we searched for CMRGs with good values of M_8, M_{16}, and M_{32} (or M_{24}, for $e > 32$). All the m_j are selected so that $r_j = (m_j^k - 1)/(m_j - 1)$ is prime, and so that the least common multiple of the $(m_j^k - 1)$ is $(m_1^k - 1) \cdots (m_J^k - 1)/2^{J-1}$ (which is the largest possible period length for the combination). In most cases, $(m_j - 1)/2$ is also prime. With these conditions, the full-period conditions are easier to satisfy and to verify, because they require (in particular) the factorization of r_j.

Table I lists some values of m and k such that m, $(m-1)/2$, and $r = (m^k-1)/(m-1)$ are all prime. These values were found by random search, using a few months of CPU time. They are useful for anyone who would like to perform additional searches for full-period MRGs.

Table I

Values of m and k such that m, $(m-1)/2$ and r are prime.

k	m		
3	$2^{31} - 21069$,	$2^{31} - 43725$,	$2^{31} - 43845$
3	$2^{32} - 209$,	$2^{32} - 22853$,	$2^{32} - 30833$
3	$2^{32} - 32969$,	$2^{32} - 33053$	
3	$2^{63} - 21129$,	$2^{63} - 275025$	
3	$2^{64} - 239669$,	$2^{64} - 525377$,	$2^{64} - 539069$
3	$2^{127} - 601821$		
3	$2^{128} - 233633$		
5	$2^{31} - 22641$,	$2^{31} - 46365$,	$2^{31} - 59601$
5	$2^{32} - 18269$,	$2^{32} - 32969$,	$2^{32} - 56789$
5	$2^{32} - 88277$,	$2^{32} - 127829$	
5	$2^{63} - 19581$,	$2^{63} - 594981$,	$2^{63} - 745281$
5	$2^{64} - 460589$,	$2^{64} - 665033$,	$2^{64} - 959417$
7	$2^{31} - 6489$,	$2^{31} - 50949$,	$2^{31} - 55341$
7	$2^{32} - 5453$,	$2^{32} - 36233$,	$2^{32} - 37277$
7	$2^{32} - 40313$,	$2^{32} - 45737$	
7	$2^{63} - 52425$,	$2^{63} - 92181$	
7	$2^{63} - 152541$,	$2^{63} - 379521$	
7	$2^{64} - 51149$,	$2^{64} - 225257$	
11	$2^{32} - 30833$,	$2^{32} - 86357$	
13	$2^{32} - 9653$,	$2^{32} - 65129$	

MRG implementations are easier and more efficient when certain constraints are imposed on the coefficients $a_{j,i}$. For example, forcing some of the coefficients to be zero save multiplications. In our search for good coefficients $a_{j,i}$, we consider also the following conditions:

(B). The product $a_{j,i}(m_j - 1)$ is less than 2^{53}.

(C). The coefficient $a_{j,i}$ satisfies $a_{j,i}(m_j \bmod a_{j,i}) < m_j$.

If Condition (B) holds, the integer $a_{j,i}x_{j,i}$ is always represented exactly in floating point on a 32-bit computer that supports the IEEE 754 floating-point arithmetic standard, with at least 53 bits of precision for the mantissa. The generator can then be implemented directly in floating-point arithmetic, which is typically faster than an integer arithmetic implementation. On the other hand, with this implementation, the state of the generator is represented over $64kJ$ bits, as opposed to $32kJ$ bits when the $x_{j,i}$ are represented as 32-bit integers. When Condition (C) is satisfied and each integer from $-m_j$ to m_j fits into a computer word, each $x_{j,i}$ can be represented as an integer over a single computer word and the product $a_{j,i}x_{j,i} \bmod m_j$ can be computed via the approximate factoring method described in Bratley, Fox, and Schrage (1987) and L'Ecuyer and Côté (1991). This condition holds if and only if $a_{j,i}^2 < m_j$ or $a_{j,i} = \lfloor m_j/z \rfloor$ for $z^2 < m_j$.

One can also force any $a_{j,i}$ to be either positive or negative. A coefficient $a_{j,i} < 0$ is equivalent to $a'_{ij} = a_{j,i} + m_j > 0$, but $|a_{j,i}|$ may satisfy a condition such as (B) or (C) that $a_{j,i} + m_j$ does not satisfy.

When (B) or (C) is imposed and some coefficients are forced to be zero, combination is usually needed for reaching good figures of merit M_T, because there is a limit on what an MRG can do with these conditions imposed on its coefficients. Combination helps because the coefficients a_j in (1) can be large even if the $a_{j,i}$ in (2) are small. To illustrate certain limitations in absence of combination, consider an MRG with a prime modulus m near 2^{32}, order $k = 7$, and for which $7 - p$ of the coefficients a_i are zero, the others being less than 2^{21} so that (B) holds. Recall (see L'Ecuyer 1997) that a general lower bound on d_t is given by

$$d_t \geq \left(1 + \sum_{i=1}^{k} a_i^2\right)^{-1/2},$$

which in our example yields $d_t \geq (1 + p(2^{21} - 1)^2)^{-1/2} \geq 1/(2^{21}\sqrt{p})$. For $t = 8$, since $\gamma_8 = \sqrt{2}$, one has $S_8 = 2^{-1/2}m^{-k/t}/d_8 < 2^{-7.5}\sqrt{p}$. With only two nonzero coefficients ($p = 2$) this gives $M_8 \leq S_8 < 1/128$, whereas if all the coefficients are nonzero ($p = 7$) this still yields $M_8 \leq S_8 < 1/68.4$. It is thus impossible to obtain a good figure of merit in this situation, for any p. Similar limitations hold if the MRG has many zero coefficients.

For several vectors $(J, k, m_1, \ldots, m_J)$ and different sets of constraints on the coefficients $a_{j,i}$, we performed random searches among the coefficients yielding maximal period length $(m_1^k - 1) \cdots (m_J^k - 1)/2^{J-1}$ for the CMRG, and retained the coefficient sets with the largest values of M_8 that we could find, those with the largest values of M_{16}, and those with the largest values of M_{32} (or M_{24} for some large m_j). The choice of $T = 8$, 16, and 32 is arbitrary. It gives generators with good lattice structures in small, medium, and large dimensions. Each random search was given a computing budget of between 20 and 40 hours of CPU time on a *SUN Sparcstation*. Performing exhaustive searches is out of the question because there are too many possibilities. The next section reports some of the results.

2. Tables of Combined MRGs with Good Figures of Merit

In the tables that we now give, a symbol * next to an M_T value means that this is the best value found for that figure of merit, within the class of CMRG considered. For each class, the m_j are fixed and the constraints (B) or (C) on the coefficients $a_{j,i}$ are given in the second column of the table. The symbol (X) means that no conditions are imposed. The coefficients not given in the tables (e.g., a_{11} and a_{22} in Table II) are equal to zero.

For example, for $J = 2$, $k = 3$, $m_1 = 2^{32} - 209$, $m_2 = 2^{32} - 22853$, $a_{11} = a_{22} = 0$, and with Condition (B) in force, the combined generator with the largest value of M_{32} that we found has $M_{32} = 0.63359$, and its coefficients are given in lines 3 and 4 from below in Table II. This generator is implemented in Figure I. Note that the generators which satisfy condition (C) in Table II also satisfy condition (B). For the values of J, k, and m_j chosen in Table II, the searches with no conditions on the coefficents did no better than those with condition (B) or (C), except for the generator in the last two lines of the table, which is marginally better with respect to M_{16} than the best one with condition (B). This means that for practical purposes, we lose nothing by imposing either (B) or (C) on the coefficients. For the larger moduli of Table III, condition (B) becomes irrelevant, and one loses very little by imposing (C). Tables IV and V give combinations of order 5 with 2 components, whereas Tables VI and VII give combinations of order 7 with 3 components. All the coefficients in Tables IV and VI satisfy (B). Condition (B+) in Table IV means that m_j times the sum of the positive coefficients $a_{j,i}$ does not exceed 2^{53}. This is slightly stronger than (B) and implies that the terms of the linear combination can be added directly in floating-point arithmetic without checking

Table II

MRGs with $J = 2$, $k = 3$, and Good Figures of Merit up to M_{32}

m_1 / m_2	Cd.	a_{12} / a_{21}	a_{13} / a_{23}	M_8	M_{16}	M_{32}
$2^{31} - 1$	B	1321911	-4129054			
$2^{31} - 21069$	B	2794761	-2188892	0.75320*	0.54812	0.54812
$2^{31} - 1$	B	3027836	-4091335			
$2^{31} - 21069$	B	4153570	-2990503	0.66216	0.65405*	0.56902
$2^{31} - 1$	B	1670453	-3445492			
$2^{31} - 21069$	B	2197254	-1967928	0.64954	0.63638	0.63442*
$2^{31} - 21069$	B	1193908	-2950125			
$2^{31} - 43725$	B	2894372	-2940180	0.75451*	0.57093	0.51825
$2^{31} - 21069$	B	1820706	-2009471			
$2^{31} - 43725$	B	1221169	-3650454	0.66209	0.65914*	0.51902
$2^{31} - 21069$	C	158402	-8405			
$2^{31} - 43725$	C	56443	-14788	0.74083*	0.55796	0.55796
$2^{31} - 21069$	C	19524	-1638034			
$2^{31} - 43725$	C	73764	-75622	0.65710	0.65292*	0.57402
$2^{31} - 21069$	C	26697	-94635			
$2^{31} - 43725$	C	17207	-32449	0.64585	0.63562	0.63257*
$2^{32} - 209$	B	1969538	-1433364			
$2^{32} - 22853$	B	847574	-739568	0.76749*	0.37869	0.37869
$2^{32} - 209$	B	1403444	-1751842			
$2^{32} - 22853$	B	2042792	-1119812	0.66825	0.65540*	0.60234
$2^{32} - 209$	B	1403580	-810728			
$2^{32} - 22853$	B	527612	-1370589	0.68561	0.63940	0.63359*
$2^{32} - 209$	X	3486492906	-835981324			
$2^{32} - 22853$	X	2107769446	-1282201325	0.66505	0.66505*	0.56803

for overflow. In Table IV, with the m_j near 2^{31}, our best combinations that satisfy (B+) are roughly as good as our best that satisfy (B). But for the m_j near 2^{32}, this is not the case: Imposing (B+) instead of (B) seems to place a limitation on S_t in dimension 6. For the combinations of order 7 with 3 components, with 3 nonzero coefficients per component, we found no good set of coefficients that satisfy (B+). We also found no good combinations in Tables V and VII for which the coefficients satisfy (C).

Table III

MRGs with $J = 2$, $k = 3$, and Good Figures of Merit up to M_{24}

m_1 / m_2	Cd.	a_{12} / a_{21} / M_8	a_{13} / a_{23} / M_{16}	M_{24}
$2^{63} - 6645$	X	2589555827131458924	-4099479422893200720	
$2^{63} - 21129$	X	3289188331138264874	-1966513844028073209	
		0.65854	0.65854*	0.49571
$2^{63} - 6645$	X	4190300628867444087	-3011960430186860296	
$2^{63} - 21129$	X	3289188331138264879	-1966513844028073209	
		0.63483	0.63483	0.63483*
$2^{63} - 6645$	C	1655695575	-6336349341	
$2^{63} - 21129$	C	31387474303	-6199136374	
		0.75429*	0.42033	0.42033
$2^{63} - 6645$	C	1671177874	-4955851730	
$2^{63} - 21129$	C	6254512935	-6964872892	
		0.69070	0.64709*	0.60405
$2^{63} - 6645$	C	1754669720	-3182104042	
$2^{63} - 21129$	C	31387477935	-6199136374	
		0.66021	0.62700	0.62700*
$2^{63} - 21129$	X	4526524762173418132	-4555864699875109770	
$2^{63} - 275025$	X	1307791354756187406	-3073682228037191328	
		0.76206*	0.48301	0.48301
$2^{63} - 21129$	X	2856694698336738094	-1298122433948874740	
$2^{63} - 275025$	X	1569635301760128104	-1851529377525193617	
		0.67365	0.67365*	0.54479
$2^{63} - 21129$	X	1526140779108535277	-2367937505303034453	
$2^{63} - 275025$	X	2780088258196613065	-3342815652037032447	
		0.63633	0.63633	0.63633*
$2^{63} - 21129$	C	3308108773	-6149300867	
$2^{63} - 275025$	C	3262668826	-7914571809	
		0.75525*	0.57777	0.51514
$2^{63} - 21129$	C	2438134156	-18272927275	
$2^{63} - 275025$	C	1675429757	-2849571296	
		0.65701	0.65328*	0.51514
$2^{63} - 21129$	C	18010381385	-5837607579	
$2^{63} - 275025$	C	3444163371	-3141078384	
		0.63477	0.63393	0.63393*

Table IV

MRGs with $J = 2$, $k = 5$, and Good Figures of Merit up to M_{32}

m_1 / m_2	Cd.	a_{12} / a_{21} / M_8	a_{14} / a_{23} / M_{16}	a_{15} / a_{25} / M_{32}
$2^{31} - 22641$	B	2727871	2605551	-2464029
$2^{31} - 46365$	B	2895584	2558064	-1854053
		0.77574*	0.50616	0.50616
$2^{31} - 22641$	B	2627540	632401	-2108408
$2^{31} - 46365$	B	2895555	2558064	-1854053
		0.66494	0.66103*	0.56765
$2^{31} - 22641$	B	2728409	760401	-3516385
$2^{31} - 46365$	B	2895587	2558063	-1854053
		0.66620	0.62885	0.62885*
$2^{31} - 22641$	B+	781863	739164	-1249628
$2^{31} - 46365$	B+	995050	1521128	-2869717
		0.76769*	0.59404	0.48965
$2^{31} - 22641$	B+	2072955	524735	-3626155
$2^{31} - 46365$	B+	839749	1782022	-1794739
		0.65267	0.65128*	0.48965
$2^{31} - 22641$	B+	343567	1162681	-1838005
$2^{31} - 46365$	B+	1358258	449185	-619098
		0.65922	0.63317	0.62644*
$2^{32} - 18269$	B	743348	1348285	-1980137
$2^{32} - 32969$	B	1788813	766578	-2064311
		0.72818*	0.52654	0.52654
$2^{32} - 18269$	B	1690742	783011	-1464677
$2^{32} - 32969$	B	1537375	1519984	-1039239
		0.65039	0.64489*	0.59046
$2^{32} - 18269$	B	1154721	1739991	-1108499
$2^{32} - 32969$	B	1776413	865203	-1641052
		0.66340	0.61130	0.61130*
$2^{32} - 18269$	B+	1033005	946785	-1387074
$2^{32} - 32969$	B+	931504	860289	-1905982
		0.57474	0.57474	0.57474*

Table V

MRGs with $J = 2$, $k = 5$, m_j near 2^{63}, and Good Figures of Merit up to M_{24}

m_1 / m_2	Cd.	a_{12} / a_{21} / M_8	a_{14} / a_{23} / M_{16}	a_{15} / a_{25} / M_{24}
$2^{63} - 19581$	X	2623120880450994287	2356691689101540791	-2787290123899037863
$2^{63} - 594981$	X	2306683785521934873	1841422677436109686	-3971733758690076701
		0.76247*	0.51720	0.51720
$2^{63} - 19581$	X	2950615467004737479	2737337638805101082	-4592181088053893523
$2^{63} - 594981$	X	4135715169147669386	4552814183224056363	-4578906048748201475
		0.68111	0.66143*	0.58368
$2^{63} - 19581$	X	1718818747424265332	3433057061817565211	-3832930842357298842
$2^{63} - 594981$	X	1774293239749025172	2893614667916396552	-4284347090005224716
		0.64397	0.64288	0.63270*

3. Implementations

Figure I gives an implementation in the C language of the CMRG given in the third entry of Table II. We call it MRG32k3a. It has 2 components of order 3, whose coefficients satisfy Condition (B). The moduli and coefficients are $m_1 = 2^{32} - 209$, $a_{11} = 0$, $a_{12} = 1403580$, $a_{13} = -810728$, $m_2 = 2^{32} - 22853$, $a_{21} = 527612$, $a_{22} = 0$, $a_{23} = -1370589$. This generator is well-behaved in all dimensions up to at least 45: In addition to $M_{32} \approx 0.6336$, one has $M_{40} \approx 0.6336$ and $M_{45} \approx 0.6225$. Its period length is $(m_1^3 - 1)(m_2^3 - 1)/2 \approx 2^{191}$. This implementation uses floating-point arithmetic and works under the (sufficient) condition that all integers between -2^{53} and 2^{53} are represented exactly in floating-point. The strings `m1`, `m2`, `a11`, etc., in the code must also be converted by the compiler to the *exact* floating-point representation of the corresponding integers (beware: the author knows compilers, for other languages than C, that do not do that correctly).

The vectors `(s10, s11, s12)` and `(s20, s21, s22)` contain the values of $(x_{1,0}, x_{1,1}, x_{1,2})$ and $(x_{2,0}, x_{2,1}, x_{2,2})$, respectively. Their initial values constitute the *seed*. Before the procedure is called for the first time, one must initialize `s10, s11, s12` to (exact) non-negative integers less than m_1 and not all zero, and `s20, s21, s22` to non-negative integers less than m_2 and not all zero. This program implements the combination (4)–(5), with $\delta_1 = -\delta_2 = 1$ and with the following slight modification: The normalization constant is $1/(m_1 + 1)$ instead of $1/m_1$, and $z_n = 0$ is converted to $z_n = m_1$. This

Table VI

MRGs with $J = 3$, $k = 7$, and Good Figures of Merit up to M_{32}

m_1 / m_2 / m_3	Cd.	a_{11} / a_{22} / a_{33} / M_8	a_{14} / a_{25} / a_{36} / M_{16}	a_{17} / a_{27} / a_{37} / M_{32}
$2^{31} - 6489$	B	4114612	695005	-1902775
$2^{31} - 50949$	B	1824834	1099113	-3119657
$2^{31} - 55341$	B	1897747	1413593	-1708684
		0.80892*	0.59276	0.59276
$2^{31} - 6489$	B	1746621	2150930	-586682
$2^{31} - 50949$	B	3047650	3229951	-741583
$2^{31} - 55341$	B	2880860	2830701	-1694599
		0.72984	0.64372*	0.58432
$2^{31} - 6489$	B	1004479	719020	-3542530
$2^{31} - 50949$	B	3259273	533655	-3434331
$2^{31} - 55341$	B	1193874	2375699	-589692
		0.70833	0.61275	0.61275*
$2^{32} - 5453$	B	1218796	1840997	-1659552
$2^{32} - 36233$	B	1581362	1977203	-963326
$2^{32} - 37277$	B	1202489	1736613	-1071212
		0.81993*	0.43817	0.43817
$2^{32} - 5453$	B	1740887	1181466	-1689373
$2^{32} - 36233$	B	1865459	1581232	-1527886
$2^{32} - 37277$	B	808720	1958655	-1081624
		0.68906	0.65815*	0.61090
$2^{32} - 5453$	B	1025652	1495670	-1555702
$2^{32} - 36233$	B	1790017	1978132	-1015534
$2^{32} - 37277$	B	1227190	1019889	-847163
		0.68699	0.64588	0.64251*

Table VII
Large MRGs with $J = 3$, $k = 7$, and Good Figures of Merit up to M_{24}

m_1		a_{11}	a_{14}	a_{17}
m_2		a_{22}	a_{25}	a_{27}
m_3	Cd.	a_{33}	a_{36}	a_{37}
		M_8	M_{16}	M_{24}
$2^{63} - 52425$	X	1199930145625658665	3713347872332282548	-4457315441628249813
$2^{63} - 92181$	X	1397544940795732264	3808491227469253752	-2779271459168535736
$2^{63} - 152541$	X	1207133271673920629	2942169185470839283	-1408095690229419395
		0.79912*	0.51149	0.51149
$2^{63} - 52425$	X	1199930145625658665	3713347872332282548	-4457315441628249813
$2^{63} - 92181$	X	3228923391594905828	1846866007242895159	-3137683670715012686
$2^{63} - 152541$	X	1987272621033941685	4562552581286095999	-2571599210827278492
		0.65741	0.64393*	0.56884
$2^{63} - 52425$	X	3066411589989614628	3773315552627701863	-2372050994168764690
$2^{63} - 92181$	X	1445357760795571378	3879290525763220258	-3915197909228525368
$2^{63} - 152541$	X	2252905204102887454	794248818025848337	-3291594373975992936
		0.66255	0.63467	0.63467*

modification is to make sure that the generator never returns exactly 0 or 1 (frequently, one takes the logarithm of u or of $1 - u$, where u is the returned value, for example to generate exponential random variables).

To implement the combination (3) instead, add:

```
#define norm2 2.328318824698632e-10
```

and replace the last two lines of the procedure by:

```
p = p1 * norm1 - p2 * norm2;
if (p < 0.0) return (p + 1.0);  else return p;
```

This would be slightly slower and may return 0.0.

This generator has been tested extensively with various empirical statistical tests and it easily passed all the tests.

Figure II provides a similar implementation, for a CMRG with two components of order 5, taken from Table IV. Its period length is $(m_1^5 - 1)(m_2^5 - 1)/2 \approx 2^{319}$. If the two components of this generator would also satisfy condition (B+), then the code could be simplified somewhat: The two lines starting with "`if (p > 0.0)`" could be removed and

```
#define norm 2.328306549295728e-10
#define m1   4294967087.0
#define m2   4294944443.0
#define a12     1403580.0
#define a13n     810728.0
#define a21      527612.0
#define a23n    1370589.0

double  s10, s11, s12, s20, s21, s22;

double MRG32k3a ()
   {
   long   k;
   double p1, p2;
   /* Component 1 */
   p1 = a12 * s11 - a13n * s10;
   k = p1 / m1;   p1 -= k * m1;   if (p1 < 0.0) p1 += m1;
   s10 = s11;   s11 = s12;   s12 = p1;
   /* Component 2 */
   p2 = a21 * s22 - a23n * s20;
   k  = p2 / m2;  p2 -= k * m2;   if (p2 < 0.0) p2 += m2;
   s20 = s21;   s21 = s22;   s22 = p2;
   /* Combination */
   if (p1 <= p2) return ((p1 - p2 + m1) * norm);
   else return ((p1 - p2) * norm);
   }
```

Figure I

A floating-point implementation in C of a 32-bit CMRG

the "`p +=`" statements that follow these lines could be incorporated with the previous line, because (B+) would guarantee that `p` could never exceed 2^{53}.

Figure III implements a generator in 64-bit integer arithmetic. It is a CMRG with 2 components of order 3, whose coefficients satisfy Condition (C) and are given in Table III. The moduli and coefficients are $m_1 = 2^{63} - 6645$, $a_{11} = 0$, $a_{12} = 1754669720$, $a_{13} = -3182104042$, $m_2 = 2^{63} - 21129$, $a_{21} = 31387477935$, $a_{22} = 0$, $a_{23} = -6199136374$. The period length is $(m_1^3 - 1)(m_2^3 - 1)/2 \approx 2^{377}$. This implementation assumes that all integers from $-m_1$ and m_1 are represented exactly in the "`long long`" type. This implementation is similar to the one given in Figure I of L'Ecuyer (1996), but with the parameters of the generator defined as constants instead of variables. This makes the code significantly faster on most computers. Again, the global variables `s10, s11, s12` (resp., `s20, s21, s22`) must be initialized to non-negative integers less than m_1 (resp., m_2) and not all zero before the first call.

To get an idea of the comparative speeds, for each generator we generated 10 million (10^7) random numbers and added them up, looked at how much CPU time (user time

```
 double  s10, s11, s12, s13, s14, s20, s21, s22, s23, s24;

 #define norm 2.328306396834613e-10
 #define m1   4294949027.0
 #define m2   4294934327.0
 #define a12     1154721.0
 #define a14     1739991.0
 #define a15n    1108499.0
 #define a21     1776413.0
 #define a23      865203.0
 #define a25n    1641052.0

double MRG32k5a ()
    {
    long   k;
    double p1, p2;
    /* Component 1 */
    p1 = a12 * s13 - a15n * s10;
    if (p1 > 0.0) p1 -= a14 * m1;
    p1 += a14 * s11;   k = p1 / m1;    p1 -= k * m1;
    if (p1 < 0.0) p1 += m1;
    s10 = s11;   s11 = s12;   s12 = s13;  s13 = s14;  s14 = p1;
    /* Component 2 */
    p2 = a21 * s24 - a25n * s20;
    if (p2 > 0.0) p2 -= a23 * m2;
    p2 += a23 * s22;    k  = p2 / m2;   p2 -= k * m2;
    if (p2 < 0.0) p2 += m2;
    s20 = s21;   s21 = s22;   s22 = s23;  s23 = s24;  s24 = p2;
    /* Combination */
    if (p1 <= p2)  return ((p1 - p2 + m1) * norm);
    else  return ((p1 - p2) * norm);
    }
```

Figure II

A floating-point implementation in C of a 32-bit CMRG of order 5 with 2 components.

```c
#define norm  1.0842021724855052e-19
#define m1    9223372036854769163
#define m2    9223372036854754679
#define a12   1754669720
#define q12   5256471877
#define r12   251304723
#define a13n  3182104042
#define q13   2898513661
#define r13   394451401
#define a21   31387477935
#define q21   293855150
#define r21   143639429
#define a23n  6199136374
#define q23   1487847900
#define r23   985240079

long long s10, s11, s12, s20, s21, s22;

double MRG63k3a ()
   {
   long long h, p12, p13, p21, p23;
   /* Component 1 */
   h = s10 / q13;   p13 = a13n * (s10 - h * q13) - h * r13;
   h = s11 / q12;   p12 = a12  * (s11 - h * q12) - h * r12;
   if (p13 < 0) p13 += m1;
   if (p12 < 0) p12 += m1 - p13;  else p12 -= p13;
   if (p12 < 0) p12 += m1;
   s10 = s11;  s11 = s12;  s12 = p12;
   /* Component 2 */
   h = s20 / q23;   p23 = a23n * (s20 - h * q23) - h * r23;
   h = s22 / q21;   p21 = a21  * (s22 - h * q21) - h * r21;
   if (p23 < 0) p23 += m2;
   if (p21 < 0) p21 += m2 - p23;  else p21 -= p23;
   if (p21 < 0) p21 += m2;
   s20 = s21;  s21 = s22;  s22 = p21;
   /* Combination */
   if (p12 > p21) return ((p12 - p21) * norm);
   else return ((p12 - p21 + m1) * norm);
   }
```

Figure III

An implementation in C, on 64-bit integers, of a CMRG of order 3 with 2 components.

Table VIII
CPU time (seconds) to generate and add 10^7 random numbers, and value of the sum

Generator	Period length $\approx$	Method	SUN Ultra-2	DEC Alpha	Sum
MRG32k3a	2^{191}	FP	5.6	8.2	5001090.95
MRG32k5a	2^{319}	FP	6.8	10.1	5000494.15
MRG63k3a	2^{377}	I	39.5	16.8	5000445.10
combMRG96a	2^{185}	I	19.8	37.6	4999897.05
combMRG96b	2^{185}	I	15.5	13.2	4999897.05
combMRG96f	2^{185}	FP	5.5	8.2	4999897.05
comblec88a	2^{61}	I	8.5	5.9	4999532.57
comblec88f	2^{61}	FP	4.2	7.9	4999532.57
drand48	2^{48}	---	20.1	8.8	---

+ system time) it took, and then printed the sum for checking purposes. This was done first on a 64-bit SUN Ultra-2 under OS 5.6, using the system's compiler (`cc`, version 4.2) with the "`-fast -xtarget=ultra -xarch=v8plusa`" options, and also on a 64-bit DEC AlphaStation 250 using the compiler `cc` at optimization level O4. The timings (in seconds) for selected generators are in Table VIII. We also indicate the period length, the type of implementation (FP for floating-point and I for integer arithmetic), and the sum of the 10^7 numbers generated. In addition to the already mentioned CMRGs, we report the timings for a C version of the 32-bit combined LCG of L'Ecuyer (1988) (`comblec88a`), the CMRG in Figure I of L'Ecuyer (1996) (`combMRG96a`), and one of the system's generators in UNIX (`drand48`). In all cases (except for `drand48`), each integer in the seed was 12345. (It is a good idea to check that your implementations reproduce the same sums.) For `comblec88a` and `combMRG96a`, the times are for the implementations in integer arithmetic as given in these papers. Implementations of these two generators in floating-point arithmetic as in Figure I are called `comblec88f` and `combMRG96f` in the table. The generator `combMRG96b` is a variant of `combMRG96a` with the moduli and multipliers defined as embedded constants in the code instead of variables as in `combMRG96a`.

Obviously, the timings depend on the type of machine. On different models of SUN computers they vary (roughly) only by a machine-dependent constant factor. On these computers, the floating-point implementation is much faster than the 32-bit integer implementation, and the implementation based on 64-bit integer arithmetic is rather slow. On the 64-bit DEC Alpha, a RISC machine with fast integer arithmetic, the implementations in integer arithmetic are more competitive. Considering the period and the quality of the lattice structure, `MRG63k3a` could be a good choice for the DEC Alpha.

The generator of Figure I gives no more than 32 bits of precision even though it returns 53-bit floating-point numbers. If more precision is desired, a simple solution uses two successive numbers produced by the generator to construct each output value. For example, if `MRG32k3a` outputs the sequence $u_1, u_2, \ldots$, one can effectively use the sequence $v_1, v_2, \ldots$ defined by $v_i = (\nu u_{2i} + u_{2i-1}) \bmod 1$ for some constant ν between 2^{-21} and 2^{-32}.

ACKNOWLEDGMENT

This work has been supported by NSERC-Canada grants No. ODGP0110050 and SMF0169893, and FCAR-Québec grant No. 93ER1654. Thanks to Anna Bragina and Richard Simard for their help in testing the code, and to Hannes Leeb, David Kelton, and two anonymous referees for their constructive comments.

REFERENCES

Bratley, P., B. L. Fox, and L. E. Schrage. 1987. *A Guide to Simulation*. Second ed. New York: Springer-Verlag.

Conway, J. H. and N. J. A. Sloane. 1988. *Sphere Packings, Lattices and Groups*. Grundlehren der Mathematischen Wissenschaften 290, New York: Springer-Verlag.

Grube, A. 1973. Mehrfach rekursiv-erzeugte Pseudo-Zufallszahlen. *Zeitschrift für angewandte Mathematik und Mechanik*, **53**, T223–T225.

Knuth, D. E. 1981. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Second ed. Reading, Mass.: Addison-Wesley.

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

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

L'Ecuyer, P. 1996. Combined multiple recursive random number generators. *Operations Research*, **44**(5), 816–822.

L'Ecuyer, P. 1997. Bad lattice structures for vectors of non-successive values produced by some linear recurrences. *INFORMS Journal on Computing*, **9**(1), 57–60.

L'Ecuyer, P. 1998a. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*. To appear.

L'Ecuyer, P. 1998b. Random number generation. In *The Handbook of Simulation*, ed. J. Banks. Wiley. To appear in Aug. 1998. Also GERAD technical report number G-96-38.

L'Ecuyer, P. 1998c. A table of linear congruential generators of different sizes and good lattice structure. *Mathematics of Computation*. To appear.

L'Ecuyer, P., J.-F. Cordeau, and R. Simard. 1997. Close-point spatial tests and their application to random number generators. Submitted.

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

L'Ecuyer, P. and R. Couture. 1997. An implementation of the lattice and spectral tests for multiple recursive linear random number generators. *INFORMS Journal on Computing*, **9**(2), 206–217.

L'Ecuyer, P. and P. Hellekalek. 1998. Random number generators: Selection criteria and testing. submitted.

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

Ripley, B. D. 1987. *Stochastic Simulation*. New York: Wiley.