Proceedings of the 2000 Winter Simulation Conference
J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, eds.

A NEW CLASS OF LINEAR FEEDBACK SHIFT REGISTER GENERATORS

Pierre L'Ecuyer
Francois Panneton

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

ABSTRACT

An efficient implementation of linear feedback shift register sequences with a given characteristic polynomial is obtained by a new method. It involves a polynomial linear congruential generator over the finite field with two elements. We obtain maximal equidistribution by constructing a suitable output mapping. Local randomness could be improved by combining the generator's output with that of some other (e.g., nonlinear and efficient) generator.

1 INTRODUCTION: LFSR GENERATORS

Linear feedback shift register (LFSR) random number generators are based on a linear recurrence of the form

$$x_n \;=\; (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod 2, \qquad (1)$$

where $k > 1$ is the *order* of the recurrence, $a_k = 1$, and $a_j \in \{0, 1\}$ for each j. This recurrence is always purely periodic (i.e., there is no transient state) and the period length of its longest cycle is $2^k - 1$ if and only if its *characteristic polynomial*

$$P(z) = -\sum_{i=0}^{k} a_i z^{k-i} \qquad (2)$$

(where $a_0 = -1$) is a primitive polynomial over $\mathbb{F}_2$, the Galois field with 2 elements (Lidl and Niederreiter 1986).

A Tausworthe-type LFSR generator (Tausworthe 1965) evolves according to (1) and produces the real number

$$u_n = \sum_{i=1}^{w} x_{nv+i-1} 2^{-i} \qquad (3)$$

at step n, where v and w are positive integers. Tezuka and L'Ecuyer (1991) and L'Ecuyer (1996) give efficient algorithms for implementing this generator when $P(z)$ is a trinomial, $P(z) = z^k - z^q - 1$, and the parameters satisfy the conditions $0 < 2q < k \le w$ and $0 < v < k - q$. Tausworthe generators that satisfy these conditions have bad statistical properties (Lindholm 1968), but combining several ones, e.g., by taking an exclusive-or of their corresponding bits x_{nv+i-1}, can provide generators with good properties (Tezuka and L'Ecuyer 1991; L'Ecuyer 1996; L'Ecuyer 1999b).

Related classes of generators are the *generalized feedback shift register* (GFSR) and *twisted GFSR* (TGFSR) generators (Tootill, Robinson, and Eagle 1973; Matsumoto and Kurita 1994; Matsumoto and Nishimura 1998; Tezuka 1995; L'Ecuyer 1994), for which each bit of the state also evolves according to a recurrence of the form (1) and where each bit of the output is a linear combination modulo 2 of the bits forming the state.

All these methods are covered by the following general linear recurrence in matrix form:

$$\mathbf{x}_n \;=\; A\mathbf{x}_{n-1}, \qquad (4)$$

$$\mathbf{y}_n \;=\; B\mathbf{x}_n, \qquad (5)$$

$$u_n \;=\; \sum_{i=1}^{w} y_{n,i-1} 2^{-i}, \qquad (6)$$

where all the operations are performed in $\mathbb{F}_2$ (i.e., modulo 2), k and w are positive integers, A is a $k \times k$ matrix, B is a $w \times k$ matrix, $\mathbf{x}_n = (x_{n,0}, \ldots, x_{n,k-1})^T$ is the k-bit *state* at step n, $\mathbf{y}_n = (y_{n,0}, \ldots, y_{n,w-1})^T$ is a w-bit vector that contains the bits of the output, and $u_n \in [0, 1)$ is the *output* at step n.

The two major considerations when choosing the general form of A and B are: (i) the statistical quality of the RNG thus obtained and (ii) the ease of constructing an efficient and portable implementation. The former is traditionally measured by the *equidistribution* of the output bits $y_{n,i}$, as recalled in Section 2. The role of the linear transformation (5) by the matrix B, also called *tempering* (Matsumoto and Kurita 1994), is precisely to improve the equidistribution (generally speaking) via some additional mixing of the bits.

In Section 3 of this paper, we introduce a class of generators allowing a fast implementation of the recurrence (4), with period length $2^k - 1$. These generators can be interpreted as linear congruential generators (LCGs) in a space of polynomials over $\mathbb{F}_2$. In Section 4, we explain how to obtain maximal equidistribution by tempering the output, i.e., by an appropriate choice of the matrix B. We discuss three specific tempering methods. In Section 5, we summarize the results of our search for good parameters. We give an example of an implementation in Section 6. In Section 7, we give an idea of the performance of this new class of generators by comparing our implementation with other known generators. A conclusion follows in Section 8.

2 EQUIDISTRIBUTION

Define Ψ_t as the set of all vectors of t successive output values produced by the generator (4–6), from all of the 2^k possible initial states. That is,

$$\Psi_t = \{\mathbf{u}_{0,t} = (u_0, \ldots, u_{t-1}) : \mathbf{x}_0 \in \mathbb{F}_2^k\}. \tag{7}$$

For a given integer $\ell \geq 0$, if we partition each axis of the unit hypercube $[0,1)^t$ into 2^ℓ equal parts, this determines a partition of the hypercube into $2^{\ell t}$ small cubes of equal volume. The point set Ψ_t (and the corresponding RNG) is called (t, ℓ)-*equidistributed*, or *t-distributed with ℓ bits of accuracy*, if each of these small cubes contains exactly $2^{k-\ell t}$ points from Ψ_t. This means that if we consider the ℓ most significant bits of the t coordinates of $\mathbf{u}_{0,t}$, the $2^{\ell t}$ different bit vectors that can be constructed appear exactly the same number of times in Ψ_t. Of course, this is possible only if $\ell t \leq k$. If Ψ_t is $\lfloor k/\ell \rfloor$-distributed with ℓ bits of accuracy for $1 \leq \ell \leq \min(k, w)$, the RNG is called *asymptotically random* or *maximally equidistributed* (ME) for the word size w (see L'Ecuyer 1996; Tezuka 1995). An ME generator has the best possible equidistribution for partitions of the unit hypercubes $[0,1)^t$ into cubic boxes of equal size, for all $\ell \leq w$ and $t\ell \leq k$. Note that every generator that uses recurrence (4) with $B = I$ and a full rank matrix A is $(1,\ \min(k, w))$-equidistributed.

To verify the equidistribution, one can write a system of linear equations that express the ℓt bits that are considered as a linear transformation of the binary vector $\mathbf{x}_0$. One has t-distribution to ℓ bits of accuracy if and only if the matrix of this linear transformation has full rank. L'Ecuyer (1996, 1999) provides tables of combined Tausworthe generators with the ME property.

3 POLYNOMIAL REPRESENTATION

Tausworthe generators can be interpreted as linear congruential generators in a space of polynomials, as follows. Let $\mathbb{F}_2[z]/(P)$ be the space of polynomials of degree less than k with coefficients in $\mathbb{F}_2$. To each state $(x_n, \ldots, x_{n+k-1})$ of the recurrence (1), we associate the polynomial

$$p_n(z) = \sum_{j=0}^{k-1} c_{n,j} z^{k-j-1} \tag{8}$$

where

$$\begin{pmatrix} c_{n,0} \\ c_{n,1} \\ \vdots \\ c_{n,k-1} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \ldots & 0 \\ a_1 & 1 & \ldots & 0 \\ \vdots & & \ddots & \vdots \\ a_{k-1} & \ldots & a_1 & 1 \end{pmatrix} \begin{pmatrix} x_n \\ x_{n+1} \\ \vdots \\ x_{n+k-1} \end{pmatrix} \bmod 2. \tag{9}$$

This mapping is obviously one-to-one, and we have (see, e.g., L'Ecuyer 1994)

$$p_{nv}(z) = z^v p_{(n-1)v}(z) \bmod P(z), \tag{10}$$

where "mod $P(z)$" means the remainder of the polynomial division by $P(z)$, with the operations on the coefficients performed in $\mathbb{F}_2$. This can be interpreted as an LCG in $\mathbb{F}_2[z]/(P)$, with modulus $P(z)$ and multiplier z^v. Note that if $P(z)$ is a primitive polynomial over $\mathbb{F}_2$, every nonzero polynomial in $\mathbb{F}_2[z]/(P)$ can be written as $z^v \bmod P(z)$ for some integer v, so there is no loss of generality in taking z^v in (10) instead of a more general polynomial.

In the past, this polynomial representation and its related *formal series* representation

$$s_n(z) = p_n(z)/P(z) = \sum_{j=1}^{\infty} x_{n+j-1} z^{-j} \tag{11}$$

have been used to *analyze* LFSR generators whose implementations were based on a state representation by the vector $(x_n, \ldots, x_{n+k-1})$ (L'Ecuyer 1994; Tezuka 1995; Couture and L'Ecuyer 2000).

In this paper, we propose to represent the state by the vector $(c_{n,0}, \ldots, c_{n,k-1})$ of coefficients of the polynomial (8). Suppose we take $v = 1$. The recurrence (10) can then be written as

$$\begin{aligned} p_n(z) &= z \sum_{j=0}^{k-1} c_{n-1,j} z^{k-j-1} \bmod P(z) \\ &= \sum_{j=0}^{k-1} c_{n-1,j} z^{k-j} \bmod P(z) \\ &= \sum_{j=0}^{k-2} c_{n-1,j+1} z^{k-j-1} + c_{n-1,0} z^k \bmod P(z) \end{aligned}$$

$$= \sum_{j=0}^{k-2} c_{n-1,j+1} z^{k-j-1} + c_{n-1,0} \sum_{j=1}^{k} a_j z^{k-j}. \qquad (12)$$

To implement (12), the coefficients $(c_{n,0}, \ldots, c_{n,k-1})$ of p_n can be stored as a k-bit string $\mathbf{c}_n$ (which can fit in one computer word if k does not exceed the word length) and the coefficients $(a_1, \ldots, a_k)$ of $P(z)$ as another k-bit string $\mathbf{a}$. To compute $p_n(z)$ from $p_{n-1}(z)$, shift $\mathbf{c}_{n-1}$ to the left by one bit, and make a bitwise exclusive-or with $\mathbf{a}$ if the original leftmost bit of $\mathbf{c}_{n-1}$ was 1. The result is $\mathbf{c}_n$. This is easy to implement and fast (especially if k does not exceed the computer's word size), for *any* characteristic polynomial. In algorithmic form, this can be written as:

$$\texttt{if } c_{n-1,0} == 1 \texttt{ then } \mathbf{c}_n = (\mathbf{c}_{n-1} \ll 1) \oplus \mathbf{a}$$
$$\texttt{else } \mathbf{c}_n = \mathbf{c}_{n-1} \ll 1$$

where $\ll s$ denotes a left shift by s bit and $\oplus$ denotes the bitwise exclusive-or operation.

This recurrence can also be written as

$$\mathbf{c}_n^T = A\mathbf{c}_{n-1}^T$$

where

$$A = \begin{pmatrix} a_1 & 1 & 0 & 0 & \ldots & 0 & 0 \\ a_2 & 0 & 1 & 0 & \ldots & 0 & 0 \\ a_3 & 0 & 0 & 1 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{k-1} & 0 & 0 & 0 & \ldots & 0 & 1 \\ a_k & 0 & 0 & 0 & \ldots & 0 & 0 \end{pmatrix}$$

This is a special case of (4) if we reinterpret $\mathbf{c}_n$ as $\mathbf{x}_n$. This form of A can provide a fast implementation and a full period. A major weakness is that there is not much "mixing of the bits" between $\mathbf{c}_{n-1}$ and $\mathbf{c}_n$. Half of the time, the bits are just shifted by one position. However, this problem can be taken care of by the tempering transformation B.

For $\nu > 1$, one can replace the matrix A by A^ν, which is equivalent to applying ν times the algorithm that computes the recurrence with $\nu = 1$. Of course, taking $\nu > 1$ will generally make the implementation slower than for $\nu = 1$.

4 LINEAR TEMPERING TRANSFORMATIONS

In this section, we discuss three convenient types of linear transformations that can be used to define the matrix B. The first one was introduced by Matsumoto and Kurita (1994). The two others are new. The main objective of applying linear transformations to the state vector is to improve the equidistribution of the random number generator. We have observed experimentally that applying only one of these linear transformations to a polynomial LCG with $\nu = 1$ is not enough to get generators with a good equidistribution. We shall therefore apply different transformations in succession.

These 3 transformations are defined by a $w \times k$ matrix B as in (5), whose first $k_w = \min(k, w)$ lines are linearly independent. This implies that B has full rank k_w, and also that the output (6) is 1-distributed to k_w bits of accuracy.

In what follows, we shall assume for simplicity that $w \leq k$. In order to be able to apply several of these transformations in succession, we will define each transformation in two steps: The first step transforms the k-dimensional vector $\mathbf{x}_n$ into a k-dimensional vector $\mathbf{z}_n = (z_{n,0}, \ldots, z_{n,k-1})$, and the second step extracts the first w bits of $\mathbf{z}_n$ to get the w-dimensional vector $\mathbf{y}_n$. When several transformations are applied in succession, this second step is performed only for the last transformation. We will denote by $\mathbf{m}_i$ a bit mask that keeps only the first i bits when applied to a given bit vector, i.e., that contains i 1's followed by zeros.

4.1 MK-Tempering

Matsumoto and Kurita (1994) have proposed a tempering defined by the following transformation, for $w \leq k$:

$$\tilde{\mathbf{y}}_n = \tilde{\mathbf{x}}_n \oplus ((\tilde{\mathbf{x}}_n \ll s_1) \,\&\, \mathbf{b}_1);$$
$$\mathbf{y}_n = \tilde{\mathbf{y}}_n \oplus ((\tilde{\mathbf{y}}_n \ll s_2) \,\&\, \mathbf{b}_2);$$

where $\tilde{\mathbf{x}}_n = (\tilde{x}_{n,0}, \ldots, \tilde{x}_{n,k_w-1})^T$ is the vector that contain the first k_w bits of $\mathbf{x}_n$, and $\&\, \mathbf{b}_i$ means a bitwise AND with some bit mask $\mathbf{b}_i$. They suggest trying values of s_2 near $k_w/2$ and s_1 near $s_2/2$. The choice of these values and of the bit masks is made so that the equidistribution of the RNG is as good as possible.

We use the following variant of this *MK tempering*:

$$\mathbf{z}_n = \mathbf{x}_n \oplus ((\mathbf{x}_n \ll s_1) \,\&\, \mathbf{b}_1);$$
$$\mathbf{z}_n = \mathbf{z}_n \oplus ((\mathbf{z}_n \ll s_2) \,\&\, \mathbf{b}_2);$$
$$\mathbf{y}_n = \mathbf{z}_n \,\&\, \mathbf{m}_w.$$

That is, we apply the tempering to the entire state $\mathbf{x}_n$, and then cut out the first w bits.

4.2 Self-Tempering

This linear transformation was originally motivated by a problem observed on the polynomial LCG of Section 3. If we use this generator directly, with $\nu = 1$, when the state $\mathbf{c}_n$ has only a few bits set to 1, then $\mathbf{c}_{n+1}$ is very likely to have only a few bits set to 1 as well. This leads to a strong dependency between the binary representations (e.g., the Hamming weights) of the successive values of $\mathbf{c}_n$ and u_n.

The *self-tempering* transformation defined here has been designed as a heuristic to reduce this dependency between the successive Hamming weights. With this transformation, the output vector $\mathbf{y}_n$ does not necessarily have fewer bits set to 1 (e.g., on the average) when the associated state vector $\mathbf{x}_n$ has only a few bits set to 1.

To define the transformation, we choose two parameters c and d so that $d < c \leq k_w$, and let $K = \lceil k/c \rceil$. This transformations cuts the bit vector $\mathbf{x}_n$ into K blocks of c bits, denoted $\mathbf{b}_1, \ldots, \mathbf{b}_K$, performs a bitwise exclusive-or of these vectors $\mathbf{b}_j$ to obtain $\mathbf{e}$, shifts $\mathbf{e}$ by d bits to the left, adds $\mathbf{e}$ to each $\mathbf{b}_j$ to obtain $\mathbf{d}_j$, and concatenates these $\mathbf{d}_j$'s to obtain $\mathbf{z}_n$. This is performed by the following steps:

1. $\mathbf{b}_j = (\mathbf{x}_n \ll c(j-1)) \,\&\, \mathbf{m}_c$ for $j = 1, \ldots, K$;
2. $\mathbf{e} = (\bigoplus_{j=1}^{K} \mathbf{b}_j) \ll d$;
3. $\mathbf{d}_j = \mathbf{b}_j \oplus \mathbf{e}$ for $j = 1, \ldots, K$;
4. $\mathbf{z}_n = \bigoplus_{j=1}^{K} (\mathbf{d}_j \gg c(j-1))$;
5. $\mathbf{y}_n = \mathbf{z}_n \,\&\, \mathbf{m}_w$.

The implementation simplifies if we take w and c equal to the computer's word length, e.g., $w = c = 32$ on a 32-bit computer. In this case, $\mathbf{x}_n$ would be represented by K 32-bit words $\mathbf{x}_n^1, \ldots, \mathbf{x}_n^K$ and the self-tempering transformation simplifies to:

1. $\mathbf{e} = (\bigoplus_{j=1}^{K} \mathbf{x}_n^j) \ll d$;
2. $\mathbf{y}_n = \mathbf{x}_n^1 \oplus \mathbf{e}$.

4.3 Permutations

This transformation permutes the coordinates by a one-to-one mapping $\pi : \mathbb{Z}_k \to \mathbb{Z}_k$. The transformation puts $z_{n,i} = x_{n,\pi(i)}$ for $i = 0, \ldots, k$. Then, $\mathbf{y}_n$ is defined by $y_{n,i} = z_{n,i}$ for $i = 0, \ldots, w$, as usual. If P is the $k \times k$ matrix that corresponds to this permutation, then $PP^T = I$ and we obtain the recurrence

$$\mathbf{z}_n = PAP^T \mathbf{z}_{n-1}. \tag{13}$$

In practice, one would choose A so that the multiplication by the matrix PAP^T is easy to implement efficiently, and one would use the recurrence (13) directly for $\mathbf{z}_n$.

In the case of a polynomial LCG, if we choose $\pi(i) = pi + q \bmod k$ where $p \neq 0$ and q are in $\mathbb{Z}_k$ and p has no common factor with k, the multiplication by PAP^T is very easy to implement. Let $\tilde{\mathbf{a}}^T = P\mathbf{a}^T$, r be such that $pr \equiv 1 \bmod k$, t be such that $pt + q \equiv 0 \bmod k$, and s be such that $ps + q \equiv k - 1 \bmod k$. The implementation becomes

```
if z_{n,t} == 1
then z_n = ((z_{n-1} <<~ r) & d_s) ⊕ ã
else z_n = (z_{n-1} <<~ r) & d_s;
```

where $\tilde{\ll} r$ means a bitwise rotation of r bits to the left and $\mathbf{d}_s$ is a mask that removes the sth bit. This can be verified by calculating the matrix PAP^T explicitly.

5 A SEARCH FOR GOOD PARAMETERS

In order to find good polynomial LCGs, we tried many primitive characteristic polynomials with different types of linear transformations. For the results reported in this paper, we have restricted our search to polynomial LCGs with characteristic polynomials of degrees 32, 64, 96, and 128, because of their ease of implementation. The equidistribution of these generators was analyzed with $w = k$, even if w will usually be less than k in a practical implementation (e.g., $w \leq 53$ if the output u_n is in double precision floating point).

To find the parameters of the linear transformations, we tried many possibilities at random. For generators with MK-tempering, we used a variant of the algorithm described by Matsumoto and Kurita (1994) to choose the values of $\mathbf{b}_1$ and $\mathbf{b}_2$.

Without tempering, all the generators tested had a very bad equidistribution in 2 dimensions or more. They were always 1-distributed to k bits of accuracy, as expected, but never 2-distributed to $k/2$ bits of accuracy. Another interesting observation is that regardless of the characteristic polynomial, they all showed the same (bad) equidistribution.

When applying only one of the linear transformations, we easily obtained generators that are $\lfloor k/\ell \rfloor$-distributed to ℓ bit of accuracy for $\ell = 1, 2, 3$, but not for $\ell > 3$. A single linear transformation of the form described in Section 4 did not suffice to "mix" the bits well enough.

When combining two linear transformations, namely permutation of the coordinates and MK-tempering, we obtained several ME generators for $k = 32$ and 64, but not for the larger values of k. We were unable to obtain generators with a characteristic polynomial of degree 96 or 128 and with satisfactory equidistribution. An example of a ME generator of period $2^{64} - 1$ is the one with characteristic polynomial $\mathbf{a}$ = `877fa931 41669185`, with permutation defined by the parameters p = 45 and q = 43, and with MK-tempering of parameters s_1 = 15, s_2 = 31, $\mathbf{b}_1$ = `77aebcea 38168000`, and $\mathbf{b}_2$ = `5f5ffec5 00000000`. (Throughout this paper, we represent the characteristic polynomials and the bit vectors $\mathbf{b}_i$ in hexadecimal notation.)

The best generator that we obtained with period $2^{96} - 1$ and with these two linear transformations is the one with characteristic polynomial $\mathbf{a}$ = `4acada15 2e647ff5 396caa79` with permutation of coordinates of parameters p = 67 and q = 55, and with MK-tempering of parameters s_1 = 23, s_2 = 47, $\mathbf{b}_1$ = `2d1dbc4f 2fa875a0 13560ba6`, and $\mathbf{b}_2$ = `3ef800b3 7b55f822 232317c7`. This generator is $\lfloor k/\ell \rfloor$-distributed to ℓ bit of accuracy for $\ell = 1, \ldots, 7, 9, 10, 11, 14, 33, \ldots, 47, 49, \ldots, 96$. For the

other values of ℓ, there is a gap of up to 3 between the upper bound $\lfloor k/\ell \rfloor$ and the dimension t for which the generator is equidistributed.

In general, the more tempering transformations we add to the output of the generators, the easier it is to obtain generators with good equidistribution. With the combination of the three types of tempering, (permutation of the coordinates, self-tempering, and MK-tempering, applied in this order), we found ME generators with characteristic polynomial of degree 96 and very good generators of degree 128 (with only one or two values of ℓ for which the generator is not $\lfloor k/\ell \rfloor$-distributed to ℓ bit accuracy). On the other hand, adding many tempering transformations slows down the generator.

An example of an ME generator with characteristic polynomial of degree 96 is given in Section 6. The best generator with period $2^{128} - 1$ was $\lfloor k/\ell \rfloor$-distributed to ℓ bit of accuracy for all $\ell = 1, \ldots, 128$, except for $\ell = 64$ where it is only 1-distributed (it is thus ME for any $w < 64$). Its characteristic polynomial is **a** = `74b480cf 73f3a60c 979782a6 787ddc13`, with permutation of the coordinates of parameters $p = 91$ and $q = 97$, followed by a self-tempering of parameters $c = 32$ and $d = 22$, and an MK-tempering of parameters $s_1 = 31$, $s_2 = 63$, $\mathbf{b}_1$ = `23d831ef 295f73be 061a1808 00000000` and $\mathbf{b}_2$ = `07edeca6 5a92f304 2e241c80 31a06893`.

Another way to improve the equidistribution is to take $\nu > 1$. We have observed that the generators are $\lfloor k/\ell \rfloor$-distributed to ℓ bit accuracy for almost all $\ell = 1, \ldots, \min(\nu, k)$. In practice, we can either apply extensive tempering to the output of a generator with $\nu = 1$, or apply less tempering to a generator with $\nu > 1$. In the first case, more tempering slows down the generator, whereas in the second case, using a larger ν usually slows down the generator as well. Further investigation should be done to seek fast implementions of the multiplication by A^ν.

6 AN EXAMPLE

Here is an example of an ME polynomial LCG with characteristic polynomial of degree $k = 96$, with **a**=`dc7348d7 18975f66 2c2ba527`. To this polynomial LCG we apply a permutation defined by $\pi(i) = 23i + 83 \bmod 96$, so that $\tilde{\mathbf{a}}$ is `4b24716e fbc6cd96 0ab7ab0c`, and the values of r, s and t are 71, 84 and 59 respectively. A self-tempering with $c = 32$ and $d = 10$ is also applied, followed by an MK tempering with $s_1 = 23$, $s_2 = 47$, $\mathbf{b}_1$ = `2fa51fb4 2e1e2000 03000000` and $\mathbf{b}_2$ = `78d849e0 55db0000 00000000`.

An implementation of this generator in language C is given in Figure 1, for $w = 32$. This generator is ME for any $w \leq k$. One can easily increase w to 53 (the maximum resolution for the `double` type) by adding `y1 * 5.421010862247e-20` to the returned value and reactivating the next-to-last instruction, which is commented out.

To implement the permutation of the coordinates, we use (13). The vector $\mathbf{z}_n$ of (13) is contained in the variables `state0`, `state1` and `state2`. These variables must be saved for the next step of the recurrence (13). The vector $\mathbf{z}_n$ resulting from each of the two other transformations is contained in the variables `y0`, `y1` and `y2`.

The state vector, stored in `state0`, `state1` and `state2`, is initialized to default values, but the initial state can be changed by changing these values (they must not be all zero) before the first call to the procedure `poly96`.

7 TIMINGS

In this section, we compare the speed of the generator `poly96` of Figure 1 to that of other random number generators. Table 1 reports the information. The number given in the table is the CPU time (in seconds) required to generate 10^7 random numbers. We used the GNU C compiler and the results are given for a Pentium-III 600 MHz processor (P-III) and an AMD Athlon 750 MHz processor (AMD), both running the Linux operating system. The generator `poly96*` is the same generator as `poly96`, but without any tempering transformation. This generator has bad equidistribution properties, as discussed in Section 5. The description of `MRG32k3a` can be found in L'Ecuyer (1999a). The generator `MT19937` was introduced in Matsumoto and Nishimura (1998) and we used the implementation provided in this reference.

Table 1: Comparaison of the Speed of Different Generators (CPU time in seconds to generate 10^7 numbers)

Generator	P-III	AMD
`MT19937`	1.73	1.20
`poly96`	2.11	1.45
`poly96*`	0.99	0.76
`MRG32k3a`	4.84	2.71

The generator `poly96*` is the fastest, about twice as fast as `poly96`, but it cannot be recommended because of its bad theoretical properties. The generator `poly96` is faster than `MRG32k3a`, but a little bit slower than `MT19937`. The generator `MT19937` also has a much larger period than `poly96`. However, it is not ME.

8 CONCLUSION

We have defined and implemented a new class of polynomial LCGs based on arithmetic modulo 2. The simplest form of this generator, with $\nu = 1$, is very fast but requires tempering transformations on the output to obtain good

```
#define a0  0x4b24716eUL
#define a1  0xfbc6cd96UL
#define a2  0xab7ab0cUL
#define B10 0x2fa51fb4UL
#define B11 0x2e1e2000UL
#define B12 0x03000000UL
#define B20 0x78d849e0UL
#define B21 0x55db0000UL
static unsigned long state0 = 0x1UL, state1 = 0x0UL, state2 = 0x0UL;

double poly96 (void) {
  unsigned long e, y0, y1, y2, w0, w1, w2;

  /* Bitwise rotation by 71 bits to the left */
  w0 = (state0 >> 25) ^ (state2 << 7);
  w1 = (state1 >> 25) ^ (state0 << 7);
  w2 = (state2 >> 25) ^ (state1 << 7);

  /* Elimination of bit 84 and verification of bit 59 */
  w2 = w2 & 0xfffff7ffUL;
  if (state1 & 0x00000010UL) {
    state0 = w0 ^ a0;  state1 = w1 ^ a1;  state2 = w2 ^ a2;
    }
  else {
    state0 = w0;  state1 = w1;  state2 = w2;
    }

  /* Self-tempering with c=32 and d=10  */
  e = (state0 ^ state1 ^ state2  ) << 10;
  y0 = state0 ^ e;  y1 = state1 ^ e;  y2 = state2 ^ e;

  /* MK-tempering with s1=23 and s2=47  */
  y0  = y0 ^ (((y1 >> 9) ^ (y0 << 23)) & B10);
  y1  = y1 ^ (((y2 >> 9) ^ (y1 << 23)) & B11);
  y2  = y2 ^ ((y2 << 23) & B12);
  y0  = y0 ^ (((y2 >> 17) ^ (y1 << 15)) & B20);
  /* y1  = y1 ^ ((y2 << 15) & B21); */
  return (y0 * 2.3283064365e-10);
  }
```

Figure 1: Implementation of poly96 in Language C

equidistribution properties. This slows down the generator to a certain extent, but it nevertheless remains competitive in terms of speed. We have experimented with some types of tempering transformations. Further exploration remains to be done in order to find more efficient transformations that provide the required equidistribution properties.

Another avenue for improving the local uniformity of the set of output vectors would be to combine a fast polynomial LCG with a small nonlinear generator, e.g., by running the two generators in parallel and adding their outputs by a bitwise exclusive-or. If these generators have relatively prime period lengths ρ_1 and ρ_2, the period length of the combination will divide (and in almost all cases, equals) the product $\rho_1\rho_2$. We plan to investigate such types of combinations, both theoretically and empirically.

ACKNOWLEDGMENTS

This work has been supported by NSERC-Canada grant No. ODGP0110050 and FCAR-Québec Grant No. 00ER3218 to the first author, and via an NSERC-Canada scholarship to the second author. Raymond Couture provided several useful ideas, especially for the material of Sections 3 and 4.3.

REFERENCES

Couture, R and P. L'Ecuyer. 2000. Lattice computations for random numbers. *Mathematics of Computation*, 69(230):757–765.

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

L'Ecuyer, P. 1996. Maximally equidistributed combined Tausworthe generators. *Mathematics of Computation*, 65(213):203–213.

L'Ecuyer, P. 1999a. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164.

L'Ecuyer, P. 1999b. Tables of maximally equidistributed combined LFSR generators. *Mathematics of Computation*, 68(225):261–269.

Lidl, R and H. Niederreiter. 1986. *Introduction to finite fields and their applications*. Cambridge: Cambridge University Press.

Lindholm, J. H. 1968. An analysis of the pseudo-randomness properties of subsequences of long *m*-sequences. *IEEE Transactions on Information Theory*, IT-14(4):569–576.

Matsumoto, M and Y. Kurita. 1994. Twisted GFSR generators II. *ACM Transactions on Modeling and Computer Simulation*, 4(3):254–266.

Matsumoto, M. and T. Nishimura. 1998. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30.

Tausworthe, R. C. 1965. Random numbers generated by linear recurrence modulo two. *Mathematics of Computation*, 19:201–209.

Tezuka, S. 1995. *Uniform random numbers: Theory and practice*. Norwell, Mass.: Kluwer Academic Publishers.

Tezuka, S and P. L'Ecuyer. 1991. Efficient and portable combined Tausworthe random number generators. *ACM Transactions on Modeling and Computer Simulation*, 1(2):99–112.

Tootill, J. P. R., W. D. Robinson, and D. J. Eagle. 1973. An asymptotically random Tausworthe sequence. *Journal of the ACM*, 20:469–481.

AUTHOR BIOGRAPHIES

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

FRANÇOIS PANNETON received a B. Sc. in computer science from the Royal Military College of Canada. He is an M.Sc. student in the "Département d'Informatique et de Recherche Opérationnelle", at the University of Montreal. He works on the design and analysis of random number generators based on linear recurrences modulo 2.