INFORMS Journal on Computing
Articles in Advance, pp. 1–12
ISSN 1091-9856 | EISSN 1526-5528

http://dx.doi.org/10.1287/ijoc.1110.0477

Large-Order Multiple Recursive Generators with Modulus $2^{31} - 1$

Lih-Yuan Deng
Department of Mathematical Sciences, University of Memphis, Memphis, Tennessee 38152,
lihdeng@memphis.edu

Jyh-Jen Horng Shiau, Henry Horng-Shing Lu
Institute of Statistics, National Chiao Tung University, Hsinchu, Taiwan, 30010, Republic of China
{jyhjen@stat.nctu.edu.tw, hslu@stat.nctu.edu.tw}

The performance of a maximum-period multiple recursive generator (MRG) depends on the choices of the recurrence order k, the prime modulus p, and the multipliers used. For a maximum-period MRG, a large-order k not only means a large period length (i.e., $p^k - 1$) but, more importantly, also guarantees the equidistribution property in high dimensions (i.e., up to k dimensions), a desirable feature for a good random-number generator. As to generating efficiency, in addition to the multipliers, some special choices of the prime modulus p can significantly speed up the generation of pseudo-random numbers by replacing the expensive modulo operation with efficient logical operations. To construct *efficient* maximum-period MRGs of a *large order*, we consider the prime modulus $p = 2^{31} - 1$ and, via extensive computer search, find two large values of k, 7,499 and 20,897, for which $p^k - 1$ can be completely factorized. The successful search is achieved with the help of some results in number theory as well as some modern factorization methods. A general class of MRGs is introduced, which includes several existing classes of efficient generators. With the factorization results, we are able to identify via computer search within this class many portable and efficient maximum-period MRGs of order 7,499 or 20,897 with prime modulus $2^{31} - 1$ and multipliers of powers-of-two decomposition. These MRGs all pass the stringent TestU01 test suite empirically.

Key words: DX/DL/DS generators; equidistribution; portable and efficient generators; Pollard's $(p-1)$ method; Pollard rho method; primality testing
History: Accepted by Marvin Nakayama, Area Editor for Simulation; received October 2009; revised June 2010, August 2010, March 2011; accepted June 2011. Published online in *Articles in Advance*.

1. Introduction

The multiple recursive generator (MRG), which is based on a kth-order linear recurrence relation with a large prime modulus p, has become increasingly popular in recent years. The performance of an MRG depends on the associated order k, the prime modulus p, and the multipliers used in the recurrence equation. The maximum-period MRGs of a large order have extremely long periods, excellent empirical performance, and more importantly, a nice property of equidistribution over high-dimensional spaces—namely, that all nonzero vectors of k values in $\{0, 1, 2, \ldots, p-1\}$ appear exactly once as k successive output values over the entire period.

In recent years, combined MRGs proposed by L'Ecuyer and his collaborators (see, for example, L'Ecuyer 1996, 1999; L'Ecuyer and Touzin 2000) have become very popular and have been implemented in commercial and public software such as MATLAB and SAS, ns2, ns3, Arena, Automod, and Witness. Among them, MRG32k3a, a combined MRG proposed in L'Ecuyer (1999), could be the most popular. By combining two maximum-period MRGs of order 3, the period length of MRG32k3a is geared up to approximately 0.49×10^{56}, a period length sufficient for most applications, and it requires only six 32-bit integers of memory space to store the states of two component MRGs. MRG32k3a is a good generator that has two large cycles and has equidistribution of order 3 if we consider the union of all cycles. For an MRG to have an equidistribution property over high dimensions, the order k needs to be large. However, the task of searching for maximum-period MRGs gets harder as the order k gets larger.

In addition to the high-order equidistribution property, the generating efficiency is another major concern; thus, we consider adopting the modulus $p = 2^{31} - 1$. With this particular form of prime modulus, we can replace the modulo operation with the much faster logical shift operation. If we further use multipliers with the powers-of-two decomposition as considered in Wu (1997), then multiplications can be

replaced as well by fast logical operations for additional savings. The main objective of this paper is to construct this kind of efficient large-order maximum-period MRGs.

The key problem in searching for maximum-period MRGs is to determine whether the corresponding kth-degree polynomial of a candidate MRG is a primitive polynomial. Alanen and Knuth (1964) and Knuth (1998) gave a set of necessary and sufficient conditions for primitive polynomials. To search for maximum-period MRGs, Deng (2004) proposed an efficient algorithm that has an early exit strategy for nonprimitive polynomials and bypasses the difficulty of factoring a large integer.

In §2, we review some common solutions for resolving the difficulties in the classical searching algorithm for large-order maximum-period MRGs. In §3, we discuss the issue of factoring large integers of the special form p^k-1 with $p=2^{31}-1$. Some classical number theory results are utilized to speed up the computer search of complete factorizations. Two large values of k, 7,499 and 2,0897, are found along with their complete factorizations of p^k-1. In §4, we describe and discuss a general class of efficient generators. We show that this new class of generators are related to three classes of portable and efficient large-order MRGs called DX/DL/DS generators, which were proposed or discussed in Deng and Xu (2003) and Deng et al. (2008b). In §5, we list some efficient and portable MRGs of a large order in this general class, which are identified by the computer search based on the searching strategy described in §2. With $p=2^{31}-1$ and $k=20{,}897$, the period length of the MRGs is extremely long, approximately $10^{195{,}009.3}$. In §6, we compare these generators with MRG32k3a in terms of their generating efficiency and other criteria. We conclude the paper in §7 with a brief summary and some remarks.

Throughout this paper, we let p be a prime number and let $\mathbb{Z}_p \equiv \{0,1,\ldots,p-1\}$ be the finite field of p elements.

2. Searching for Maximum-Period MRGs

MRGs have become one of the most commonly used random number generators in computer simulations. Sequences of pseudo-random numbers can be generated by an MRG of order k via the following kth-order linear recurrence equation:

$$X_i = (\alpha_1 X_{i-1} + \cdots + \alpha_k X_{i-k}) \bmod p, \quad i \geq k, \qquad (1)$$

with any not-all-zero initial seeds $(X_0,\ldots,X_{k-1})$. Here, the modulus p is a large prime number. Since $X_i \in \mathbb{Z}_p$, X_i can be easily transformed to a $U_i \in [0,1)$, $U_i \in [0,1]$, or $U_i \in (0,1)$ by letting $U_i = X_i/p$, $U_i = X_i/(p-1)$, or $U_i = (X_i+0.5)/p$, respectively.

It is well known that the maximum period of an MRG is p^k-1, which is achieved if and only if its characteristic polynomial

$$f(x) = x^k - \alpha_1 x^{k-1} - \cdots - \alpha_k \qquad (2)$$

is a primitive polynomial. Alanen and Knuth (1964) gave three conditions for verifying the primitivity of $f(x)$. See also Knuth (1998).

In addition to the long period, a maximum-period MRG is known to have the nice property of equidistribution up to k dimensions: every t-tuple $(1 \leq t \leq k)$ of integers between 0 and $p-1$ appears exactly the same number of times (p^{k-t}) over its entire period (p^k-1), with the exception of the all-zero tuple that appears one time less $(p^{k-t}-1)$. See, for example, Lidl and Niederreiter (1994, Theorem 7.43).

2.1. Resolving Difficulties in Finding Maximum-Period MRGs

As mentioned in §1, when k and p are large, it can be time consuming to determine whether $f(x)$ in (2) is a primitive polynomial when checking the conditions of Alanen and Knuth (1964) directly. To speed up the search, Deng (2004) proposed an early exit strategy when $f(x)$ is not a primitive polynomial. Another difficulty is with regard to the factorization of p^k-1, which can be very hard when k and p are large. Given the current technology, factoring an integer with 200 digits (or more) is hard unless this integer has a special structure. Alternatively, L'Ecuyer et al. (1993) suggested finding a prime p for a given k such that

$$R(k,p) = (p^k-1)/(p-1) \qquad (3)$$

is also a prime number, to bypass the difficulty of factorization. Such $R(k,p)$ was termed a *generalized Mersenne prime* (*GMP*) in Deng (2004), where a list of GMPs with k up to 1,511 was also given. Later, Deng (2008) found some additional GMPs for order k up to 10,007.

For achieving computational efficiency in generating random numbers, in this paper, we consider adopting $p=2^{31}-1$. With this p, the original thought was to find k such that $R(k,p)$ is a prime number. Unfortunately, Deng (2005) reported that the search was unsuccessful for k up to 25,000. We conducted another search for k further up to 60,000, which was also unsuccessful. The computation was quite intensive. Specifically, for each candidate k, we checked whether $R(k,p)$ is a "probable" prime using some probabilistic tests, which can be highly efficient, and the test result is definite when a number is declared composite. The required computing time for each candidate k is more than 10 hours on a PC with 2.8 GHz

CPU when k reaches 60,000. Moreover, to the best of our knowledge, there are no known mathematical results about the existence of prime $R(k,p)$ for $p=2^{31}-1$. Therefore, in this study, instead of finding a prime $R(k,p)$ to bypass complete factorization, we switch to the strategy of finding k such that $R(k,p)$ in (3) is *easy* to be factored.

2.2. Efficient MRGs with the Prime Modulus $p=2^{31}-1$

In this paper, we focus on finding maximum-period MRGs in (1) with $p=2^{31}-1$, a popular modulus with computational advantages. For example, the expensive modulo operation $(\bmod p)$ can be replaced by much faster logical operations. Specifically, Deng and Xu (2003) defined a simple C function to compute $z \bmod 2^{31}-1$ as

```
unsigned long MODP(unsigned long z)
    {return (((z)&p)+((z)>>31));},
```

where z is the operand, "&" is the "bitwise logical and" operation, and ">>" is the logical right shift operation. Also, using multipliers of the form $2^r \pm 2^w$, called the powers-of-two decomposition, can result in fast computation by using only shift and addition operations. The utilization of the powers-of-two decomposition was suggested by Wu (1997) for linear congruential generators (LCGs) with prime moduli of the form 2^q-1. L'Ecuyer and Simard (1999) generalized the method to the case when the prime modulus has the form 2^q-h for small h. However, L'Ecuyer and Simard (1999) also pointed out that such LCGs have bad statistical properties because the recurrence does not "mix the bits" well enough. On the other hand, Deng (2005) and Deng et al. (2008a) found that similar problems do not occur in many large-order MRGs with $p=2^{31}-1$. The latter also conducted a speed comparison on several generators and reported that a DX generator with a powers-of-two multiplier is about twice as fast as a DX generator with a general multiplier. DX generators are MRGs of a special form proposed by Deng and Xu (2003) for efficiency. See §4.2 for a brief description.

With the prime modulus p fixed as $2^{31}-1$, factoring p^k-1 can be quite difficult in general. In the following, we discuss the possibility of getting complete factorizations for some large values of k for which p^k-1 has some special structures.

In the next section, we discuss some useful techniques and theories in general for factoring a large integer and more specifically for factoring p^k-1.

3. Factoring p^k-1 When $p=2^{31}-1$

It is hard to find the complete factorization of $R(k,p)$ for a general k. However, if there are some special structures on $R(k,p)$ such that certain powerful factorization methods can be utilized, then it may become easier to find *some* ks with the complete factorization of $R(k,p)$.

In this section, we explore three approaches to finding k and the complete factorization of $R(k,p)=(p^k-1)/(p-1)$ with $p=2^{31}-1$. Because each approach is more suitable for a certain type of $R(k,p)$, we use it to search over those ks with this particular type of $R(k,p)$ for complete factorization.

3.1. Factorization of (p^k-1) via Trial Divisions

The first approach is to find k for which $R(k,p)$ is easy to factor because it contains some "small" prime factors (say, $\le 10^9$) and only one single huge prime factor. Specifically, Deng (2005) found three such ks (i.e., $k=47$, 643, and 1,597) by considering $R(k,p)=q\times H$, where q is a prime not greater than 10^9 and H is a huge prime.

In this paper, for large values of k, we conduct a similar search as in Deng (2005) by removing all small prime factors less than a chosen value Δ. Specifically, let the factorization of $R(k,p)$ be written as

$$R(k,p)=\prod_{i=1}^{r} q_i^{c_i} \quad \text{for some } c_i \ge 1,\ i=1,\ldots,r, \tag{4}$$

where $q_1<q_2<\cdots<q_r=H_k$ are (mostly) unknown primes. If H_k (the largest prime factor of $R(k,p)$) is the only prime factor greater than Δ, then we can find the complete factorization of $R(k,p)$ by performing trial divisions up to Δ. Whereas a larger value of Δ can increase the possibility of finding all small factors, it also increases the computing time required for finding all prime factors less than Δ. In this study, following Deng (2005), we set $\Delta=10^9$.

When k is large, computation using trivial trial division to search for small prime factors of $R(k,p)$ can be very time consuming. Fortunately, the search can be greatly sped up with the help of Legendre's theorem, given below.

Theorem 1 (Theorem 5.7 in Riesel 1994, p. 165). *Let k be a prime, and let a,b be integers with $\gcd(a,b)=1$. For every prime factor q of $(a^k-b^k)/(a-b)$, we have $q=1 \bmod 2k$.*

By Theorem 1, any prime factor q of $R(k,p)$ has the form

$$q=2kC+1 \tag{5}$$

for some positive integer C. Therefore, we can compute Q, which is the product of all primes $q<10^9$ (say) of the form $2kC+1$. We then compute

$$D=\gcd(R(k,p),Q)$$

and apply some probabilistic primality tests to $R(k,p)/D$.

Having performed this procedure for $k < 60{,}000$, we find $k = 7{,}499$, for which

$$R(7{,}499, p) = 13{,}333{,}223 \times 26{,}606{,}453 \times 60{,}306{,}959 \times 296{,}465{,}467 \times H_{7{,}499}, \quad (6)$$

where $H_{7,499}$ is a probable prime. We then verify the primality of $H_{7,499}$ with the probabilistic tests provided by commercial packages such as Maple and Mathematica. We further perform the *industrial prime test* proposed in Damgård et al. (1993) on $H_{7,499}$. It passes both tests. See also Algorithm 3.4.7 in Crandall and Pomerance (2000, p. 126) for the *industrial prime test*. As discussed in Crandall and Pomerance (2000), the probability of committing a false-positive error can be controlled to a level much smaller than 10^{-200}. This error probability is much smaller than the computer software error or the hardware error. Therefore, according to Crandall and Pomerance (2000, p. 127), $H_{7,499}$ can be safely accepted as a "prime" for all but the most sensitive practical applications.

3.2. Factorization of $(p^k - 1)$ via Pollard's $(p-1)$ Method

The second approach is to use some modern factorization methods to "quickly" find some large factors of $R(k, p)$. We first describe one popular method called Pollard's $(p-1)$ method (Pollard 1974). This method is most useful when every prime factor q (except the largest one) of $R(k, p)$ in (4) has the property that $q-1$ has only "small" prime factors. To be more specific, if, for each prime factor $q \neq q_r$ of $R(k, p)$, there exists a reasonably small upper bound W such that

$$w \leq W \quad (7)$$

for all prime w satisfying $w \mid (q-1)$, then Pollard's $(p-1)$ method can find the complete factorization of $R(k, p)$ as described below.

To simplify the notation, let q be any prime factor q_i $(i < r)$ of $R(k, p)$ in (4), and denote $R(k, p)$ by R. If we can find a Q such that $(q-1) \mid Q$, then Fermat's little theorem, which states that $A^{q-1} = 1 \bmod q$ for any $A \neq 0 \bmod q$, tells us that

$$A^Q = 1 \bmod q,$$

provided that $A \neq 0 \bmod q$. Since q is a common factor of R and $(A^Q - 1 \bmod R)$, we may be able to find a factor of R by computing

$$\gcd(A^Q - 1, R)$$

with an appropriately chosen Q.

To implement Pollard's $(p-1)$ method, we need to set the upper bound W in (7) and then find Q accordingly. In this study, we follow a reasonable choice of Q described in Riesel (1994) when W is given. That is, let

$$Q = \prod_{q \in S_W} q^c,$$

where the set S_W contains all primes q smaller than W, and $c \geq 1$ is the integer associated with q such that $q^c \leq W < q^{c+1}$.

Now we only need to choose the upper bound W. According to Theorem 1, for any prime factor q of R, k is a prime factor of $q-1$. Thus, by (7), the smallest potential value for the upper bound W is k. Therefore, we need to choose $W \geq k$ for Pollard's $(p-1)$ method.

In this study, we choose $W = k$ and $A = 2$ and search over various values of k. The above-described procedure gives $k = 20{,}897$ and

$$R(20{,}897, p) = 5{,}558{,}603 \times 110{,}346{,}315{,}943 \times H_{20{,}897}, \quad (8)$$

where $H_{20,897}$ is a huge probable prime tested by the procedure described in §3.1.

When k is very large, the computing time for the above procedure can be long. In this case, we can consider a smaller value of $W < k$ in (7) with $A = 2^k$. As an illustration, for $k = 20{,}897$, the complete factorization (8) gives

$$q_1 - 1 = 5{,}558{,}602 = 2 \times 7 \times 19 \times 20{,}897,$$
$$q_2 - 1 = 110{,}346{,}315{,}942 = 2 \times 3 \times 43 \times 97 \times \mathbf{211} \times 20{,}897;$$

for $k = 7{,}499$, the complete factorization (6) gives

$$q_1 - 1 = 13{,}333{,}222 = 2 \times 7 \times 127 \times 7{,}499,$$
$$q_2 - 1 = 26{,}606{,}452 = 2^2 \times 887 \times 7{,}499,$$
$$q_3 - 1 = 60{,}306{,}958 = 2 \times \mathbf{4{,}021} \times 7{,}499,$$
$$q_4 - 1 = 296{,}465{,}466 = 2 \times 3 \times 11 \times 599 \times 7{,}499.$$

Note that if we could foresee this, then we can set the values of W in (7) with $A = 2^k$ for $k = 7{,}499$ and $k = 20{,}897$ as small as 4,021 and 211 (marked in boldface above), respectively, to find all "small" factors of the corresponding $R(k, p)$. Unfortunately, there is no way we could know these smallest bounds in advance. Thus choosing a value for W larger than required is still a good strategy for Pollard's $(p-1)$ method.

3.3. Factorization of $(p^k - 1)$ via Pollard's rho Method

Pollard's rho method is another powerful factorization method (Pollard 1975). This method does not assume special structures such as those for the two methods described above on the integer to factor. However, successful searches are not guaranteed because it is a probabilistic method, and the number of iterations required (for a successful search) may be

larger than the upper bound set in the search algorithm. We use this method to find a factor q of R according to the algorithm described below.

Let v_0 be a chosen initial seed, and let $h(\cdot)$ be a function modulo R. Choices of $h(\cdot)$ will be discussed later.

Step 1. Set the initial values $x_0 = v_0$ and $y_0 = v_0$. Generate two sequences $\{x_i,\ i \geq 0\}$ and $\{y_i,\ i \geq 0\}$ by

$$x_i = h(x_{i-1}) \quad \text{and} \quad y_i = h(h(y_{i-1})).$$

Note that $\{y_i,\ i \geq 0\}$ is simply $\{x_{2i},\ i \geq 0\}$. This key step is based on Floyd's cycle-finding algorithm as discussed in Riesel (1994, p. 178).

Step 2. If $D \equiv \gcd(|x_i - y_i|, R) = 1$, then move on to the next i; else, if $D < R$, return D as a factor of R. Otherwise (i.e., $D = R$, which means either R is a prime or Pollard's rho algorithm has failed), retry with another initial value v_0 when R is not a prime.

As explained in Riesel (1994, pp. 185–186), for a general factorization problem, it is common to choose $h(x) = x^2 + a \bmod R$. However, because all the prime factors of R are of the form $2kC+1$, using $h(x) = x^k + a \bmod R$ is likely to reduce the number of iterations needed to discover a prime factor q by a proportion of $\sqrt{k-1}$ when compared to using $h(x) = x^2 + a \bmod R$, the common choice. It is interesting to observe that, with $h(x) = x^k + a \bmod R$, we have

$$\begin{aligned} x_i - x_j &= x_{i-1}^k - x_{j-1}^k \\ &= (x_{i-1} - x_{j-1})\left(\frac{x_{i-1}^k - x_{j-1}^k}{x_{i-1} - x_{j-1}}\right) \quad \text{for } i \neq j. \end{aligned}$$

Applying the above formula iteratively, we have

$$x_i - x_j = \left(\frac{x_{i-1}^k - x_{j-1}^k}{x_{i-1} - x_{j-1}}\right)\left(\frac{x_{i-2}^k - x_{j-2}^k}{x_{i-2} - x_{j-2}}\right)\left(\frac{x_{i-3}^k - x_{j-3}^k}{x_{i-3} - x_{j-3}}\right)\cdots.$$

According to Theorem 1, each term of the above product may contain prime factors of the form $2kC+1$. Thus, the iterative steps in Pollard's rho method can accumulate factors of this type.

By using Pollard's rho method, we confirm the finding of all the prime factors of $R(k,p)$ for both $k = 7{,}499$ and $k = 20{,}897$. The number of iterations needed to find all "small" factors appears random for each try. For $k = 7{,}499$, Pollard's rho method finds all the factors in less than 350 iterations. For $k = 20{,}897$, the number of iterations needed ranges from 208 to 2,607 with various initial values. We have designed a Maple program to implement the aforementioned algorithm. The required computing time is not reported here because it highly depends on the (random) number of iterations needed and the computing hardware, software (other packages or versions), and/or operating system used.

3.4. Discussion

It is possible to utilize more powerful modern factorization algorithms that can find some additional large factors of R and then hopefully find the complete factorization. For example, Lenstra's elliptic curve method (Lenstra 1987) and the Number field sieve method (Pollard 1993) are two powerful factorization methods. Both methods have found prime factors of 50 digits or more for some cases as reported in the literature. See Riesel (1994) for more details. However, we were unsuccessful in finding factorizations with larger values of k using these methods.

We remark that, although the two $R(k,p)$s mentioned in this section are big numbers with many thousands of digits, they are easily handled by some popular commercial symbolic language packages such as Maple or Mathematica without any problems on number representation. We can also use some free and popular multiprecision software packages such as Victor Shoup's NTL (http://www.shoup.net/ntl/) or GMP (http://gmplib.org/). In terms of multiprecision computation, NTL and GMP are more efficient than Maple or Mathematica.

4. Efficient and Portable Large-Order MRGs

4.1. A General Class of Efficient Generators

In general, an MRG is computationally efficient if either its recurrence Equation (1) has very few nonzero terms or it can be implemented efficiently with a higher-order recurrence equation of very few nonzero terms. A DX generator, proposed by Deng and Xu (2003) and Deng (2005), is an example of the former case. DL and DS generators considered in Deng et al. (2008b), which have many nonzero terms with the same nonzero coefficient in the recurrence equation, are two examples of the latter case.

We now consider a more general class of generators as follows. For an integer C, let $S_C = \{j \mid \alpha_j = C\}$ be the set of the indices j with $\alpha_j = C$. Consider the following class of generators:

$$\begin{aligned} X_i - AX_{i-g} = B\Bigg(\sum_{j\in S_B} X_{i-j}\Bigg) \\ + D\Bigg(\sum_{j\in S_D} X_{i-j}\Bigg) \bmod p, \quad i \geq k, \end{aligned} \tag{9}$$

where A, B, D, and g are integers appropriately chosen.

This general class of generators has several interesting special cases, including DX, DL, DS, and other types of generators, as discussed in brief next.

4.2. DX Generators

The DX generators comprise a system of portable, efficient, and maximum-period MRGs in which the coefficients of the nonzero multipliers are the same. In particular, DX-*k*-*s*-*t* generators, as considered in Deng (2005), can be obtained from (9) with $A=0$ as follows:

1. DX-*k*-1-*t*: $D=1$, $S_1=\{t\}$, $S_B=\{k\}$,
2. DX-*k*-2-*t*: $S_D=\varnothing$, $S_B=\{t,k\}$,
3. DX-*k*-3-*t*: $S_D=\varnothing$, $S_B=\{t,\lceil k/2\rceil,k\}$,
4. DX-*k*-4-*t*: $S_D=\varnothing$, $S_B=\{t,\lceil k/3\rceil,\lceil 2k/3\rceil,k\}$,

where s is the number of terms with coefficient B, and t is the first index for which $\alpha_j\neq 0$.

4.3. DL and DS Generators

DL/DS generators can be obtained from (9) with some special values for the parameters as in the following:

1. DL-*k*-*t*:

$$X_i=B(X_{i-t}+X_{i-t-1}+\cdots+X_{i-k}) \bmod p, \quad i\geq k. \tag{10}$$

Such DL generators can be implemented efficiently by

$$X_i=X_{i-1}+B(X_{i-t}-X_{i-(k+1)}) \bmod p, \quad i\geq k+1. \tag{11}$$

Thus, DL-*k*-*t* generators can be considered a special case of (9) with $A=g=1$, $D=-B$, $S_{-B}=\{k+1\}$, and $S_B=\{t\}$. For simplicity, the default case of $t=1$ is referred to as the DL-*k* generators.

2. DS-*k*-*t*: Deng et al. (2008b) considered another class of generators with many nonzero coefficients, called DS generators:

$$X_i=B\sum_{j=1,\,j\neq t}^{k} X_{i-j} \bmod p, \tag{12}$$

which can be efficiently implemented by

$$X_i=X_{i-1}+B(X_{i-1}-X_{i-t}+X_{i-t-1}-X_{i-k-1}) \bmod p, \quad i\geq k+1. \tag{13}$$

The parameter t of the zero-coefficient index can be chosen arbitrarily. DS-*k*-*t* generators can be considered a special case of (9) with $A=g=1$, $D=-B$, $S_{-B}=\{t,k+1\}$, and $S_B=\{1,t+1\}$. We refer to the default case of $t=\lceil k/2\rceil$ as the DS-*k* generators.

4.4. Advantages of the New Class of Generators

Because of the limited choices of r and w for the multiplier $B=2^r\pm 2^w$, and by restricting the modulus to $p=2^{31}-1$, the default-case maximum-period DX-*k*-*s*, DL-*k*, and DS-*k* generators may not exist when the order k is large. Hence, it is necessary to search over a more general class of generators, and that was the main motivation for introducing the additional parameter t in DX, DL, and DS generators in the previous works.

On the other hand, introducing AX_{i-g} in the new class of generators (9) has several advantages. First, it provides a general form to cover DX generators and DL/DS generators by letting $A=0$ and $A=g=1$, respectively. When $A=0$, the recurrence Equation (9) computes a linear combination of few nonzero terms. When $A=g=1$, the successive difference X_i-X_{i-1} can be expressed as a linear combination of just a few nonzero terms. Second, similar to the role of the parameter t, it can greatly expand the searching space of generators by varying values of A and/or g when the form of the multiplier is restricted. The last and the most important advantage can be described by the results of some empirical studies performed on DX-*k*-*s*-*t* generators. We have observed that when t gets larger, the empirical performance of DX-*k*-*s*-*t* generators becomes poorer. Usually, we fix $t=1$ to search for the multiplier B to achieve the maximum period. However, if we restrict $B=2^r\pm 2^w$ for some r, w, then t may need to be large to obtain a successful search for a DX-*k*-*s*-*t* generator, especially when the order k is large. For example, we have found a DX-*k*-*s*-*t* generator (with $k=20{,}897$, $s=1$, $t=382$, and $B=134{,}217{,}736=2^{27}+2^3$) consistently fails one particular test ("SerialOver, with dimension 2") in the Crush battery of TestU01 test suite, a popular package developed by Pierre L'Ecuyer for testing random number generators. See L'Ecuyer and Simard (2007) for more details about TestU01. The same failure also occurs when we replace the powers-of-two multiplier ($B=2^r\pm 2^w$) with a general B while $t=382$ is fixed. It would be interesting to know the reason why DX-*k*-*s*-*t* generators fail for a large value of t—say, $t>300$. According to our observations, this problem appears to be more profound when $s=1$ and t is very large. Thus, instead of searching over t to find DX-*k*-*s*-*t* generators, we propose a modification of the original DX-*k*-*s* generators by adding X_{i-g} in the recurrence as follows:

1. DX*-*k*-1-*g*:

$$X_i-X_{i-g}=X_{i-1}+BX_{i-k} \bmod p, \quad i\geq k. \tag{14}$$

2. DX*-*k*-2-*g*:

$$X_i-X_{i-g}=B(X_{i-1}+X_{i-k}) \bmod p, \quad i\geq k. \tag{15}$$

3. DX*-*k*-3-*g*:

$$X_i-X_{i-g}=B(X_{i-1}+X_{i-\lceil k/2\rceil}+X_{i-k}) \bmod p, \quad i\geq k. \tag{16}$$

4. DX*-*k*-4-*g*:

$$X_i-X_{i-g}=B(X_{i-1}+X_{i-\lceil k/3\rceil}+X_{i-\lceil 2k/3\rceil}+X_{i-k}) \bmod p, \quad i\geq k. \tag{17}$$

Note that the DX*-*k*-*s*-*g* generators given in (14)–(17) are simply special cases of (9) with $A=1$. To search for maximum-period DX* generators with $B=2^r\pm 2^w$, we can then try various values of g. Our evaluation indicates that the corresponding DX* generators do pass the extensive empirical tests provided in the Crush battery of TestU01.

Computing codes in C for implementing DX generators for any order k have been provided and discussed in Deng (2005), and they are available at http://www.cs.memphis.edu/~dengl/dx-rng/. Codes for the other generators described above are available from the authors upon request.

5. Tables of Large-Order DX, DL, and DS Generators

In §3, we have presented the complete factorization of $R(k, 2^{31}-1)$ for $k=7{,}499$ and $k=20{,}897$. Using the efficient searching algorithm described in Deng (2004, 2005), it is straightforward to find maximum-period DX, DL, and DS generators. The period lengths of these generators are approximately $10^{69{,}980.1}$ and $10^{195{,}009.3}$ for $k=7{,}499$ and $k=20{,}897$, respectively.

In this section, we tabulate three kinds of large-order MRGs found via computer search: (i) DX, DL, and DS generators with $p=2^{31}-1$ and general B; (ii) DX, DL, and DS generators with $p=2^{31}-1$ and $B=2^r\pm 2^w$; and (iii) DX* generators with $p=2^{31}-1$ and $B=2^r\pm 2^w$.

5.1. DX, DL, and DS Generators with $p=2^{31}-1$ and General B

We search for maximum-period MRGs under the following conditions.

Condition 1 ($\min B$). We start the search of B from the smallest value and move upward until a maximum-period MRG of the specified type (DX/DL/DS) is found. As discussed in Deng and Xu (2003) and Deng (2005), the generators obtained under this condition are not recommended for general use because B is too small. On the other hand, the small magnitude makes such B simpler to implement in a procedure developed for automatic generation of other maximum-period MRGs from a given MRG, as discussed in Deng et al. (2009). In addition, they all pass the stringent empirical tests in the Crush battery of TestU01.

Condition 2 ($B<2^e$). We start the search of B from the upper bound 2^e for some e and move downward. The choice of e depends on the type of MRGs under search. For an exact computation using the IEEE double-precision standard, we choose $e=20$ for DX generators with $s=1$, 2, and DL generators, and we choose $e=19$ for DX generators with $s=3$, 4, and DS generators.

Condition 3 ($B<2^{30}$). We start the search of B from the upper bound 2^{30} and move downward. The size of B is large, and it would be suitable when a 64-bit integer type is available in the computing platform and the compiler. However, without using 64-bit data types/operations, a portable MRG can be implemented in 32-bit operations at the expense of slight generating inefficiency as follows. Following Deng (2005), we find the smallest positive integer u such that

$$B+u\times p=C_1\times C_2 \quad \text{for some } 0<C_1,\ C_2<2^{19}. \tag{18}$$

Therefore, we find

$$B=C_1\times C_2 \bmod p.$$

This method is more general than the idea of finding a multiplier B such that it is exactly equal to the product of two small numbers, as considered in Marse and Roberts (1983).

Because the range of B is wide, we can fix the parameter t as its default value for the generators considered. That is, $t=1$ for DX and DL generators, and $t=\lceil k/2\rceil$ for DS generators.

For $k=7{,}499$ and $k=20{,}897$, some maximum-period DX, DL, and DS generators are found and listed in Table 1. We remark that the required searching time is clearly random, and it increases significantly as we move from $k=7{,}499$ to $k=20{,}897$. The required searching time ranges from a few hours to two to three days and from three days to more than two weeks for $k=7{,}499$ and $k=20{,}897$, respectively.

5.2. DX, DL, and DS Generators with $p=2^{31}-1$ and $B=2^r\pm 2^w$

As mentioned earlier, we can further increase the generating speed by considering $B=2^r\pm 2^w$, where

Table 1 **List of DX-*k*-*s*, DL-*k*, and DS-*k* with Various Sizes of *B* for $k=7{,}499$ and $k=20{,}897$**

Generator	$\min B$	$B<2^e$	$B<2^{30}$	u	C_1	C_2
			$k=7{,}499$			
DX($s=1$)	13,620	967,501	1,073,735,056	1	28,979	111,157
DX($s=2$)	18,178	1,038,757	1,073,706,686	0	8,999	119,314
DX($s=3$)	2,307	517,486	1,073,741,559	1	39,878	80,777
DX($s=4$)	25,972	519,708	1,073,723,713	0	4,273	251,281
DL	38,999	1,035,347	1,073,716,921	1	19,304	166,867
DS	26,908	451,111	1,073,731,005	1	7,043	457,364
			$k=20{,}897$			
DX($s=1$)	29,260	1,009,278	1,073,616,009	3	28,350	265,117
DX($s=2$)	45,072	1,028,880	1,073,738,158	0	5,554	193,327
DX($s=3$)	10,706	490,124	1,073,714,805	0	22,531	47,655
DX($s=4$)	110,120	514,809	1,073,718,732	2	67,034	80,089
DL	97,155	972,308	1,073,721,537	0	2,331	460,627
DS	33,948	439,186	1,073,656,108	0	26,908	39,901

Table 2 **List of DX-*k*-*s*-*t*, DL-*k*-*t*, and DS-*k*-*t* with $B=2^r+2^w$ for $k=7{,}499$ and $k=20{,}897$**

Generator	t	B	(r,w)
		$k=7{,}499$	
DX($s=1$)	29	1,048,832	(20, 8)
DX($s=2$)	64	537,001,984	(29, 17)
DX($s=3$)	70	134,479,872	(27, 18)
DX($s=4$)	11	1,048,578	(20, 1)
DL	13	2,097,280	(21, 7)
DL	125	2,097,156	(21, 2)
DS	3,915	1,050,624	(20, 11)
DS	3,754	1,048,832	(20, 8)
		$k=20{,}897$	
DX($s=1$)	23	1,073,750,016	(30, 13)
DX($s=2$)	95	4,198,400	(22, 12)
DX($s=3$)	63	33,554,440	(25, 3)
DX($s=4$)	148	268,435,968	(28, 9)
DL	432	524,289	(19, 0)
DL	536	525,312	(19, 10)
DL	676	1,049,600	(20, 10)
DL	456	2,097,156	(21, 2)
DS	11,050	1,056,768	(20, 13)
DS	10,661	67,633,152	(26, 19)
DS	11,270	33,554,496	(25, 6)
DS	11,290	16,793,600	(24, 14)
DS	11,200	33,619,968	(25, 16)

Table 3 **List of DX*-*k*-*s*-*g* Generators with $B=2^r+2^w$ for $k=7{,}499$ and $k=20{,}897$**

DX* generator	g	B	(r,w)
		$k=7{,}499$	
DX*($s=1$)	45	134,217,984	(27, 8)
DX*($s=1$)	193	8,388,612	(23, 2)
DX*($s=1$)	330	8,388,672	(23, 6)
DX*($s=1$)	349	2,113,536	(21, 14)
DX*($s=1$)	360	528,384	(19, 12)
DX*($s=1$)	376	262,146	(18, 1)
DX*($s=1$)	383	1,074,003,968	(30, 18)
DX*($s=2$)	17	134,217,792	(27, 6)
DX*($s=2$)	193	16,908,288	(24, 17)
DX*($s=2$)	221	2,097,156	(21, 2)
DX*($s=2$)	222	536,870,944	(29, 5)
DX*($s=2$)	257	536,871,040	(29, 7)
DX*($s=3$)	197	541,065,216	(29, 22)
DX*($s=3$)	257	4,198,400	(22, 12)
DX*($s=3$)	496	268,500,992	(28, 16)
DX*($s=4$)	131	536,871,040	(29, 7)
DX*($s=4$)	69	67,633,152	(26, 19)
DX*($s=4$)	345	1,074,790,400	(30, 20)
		$k=20{,}897$	
DX*($s=1$)	53	1,074,790,400	(30, 20)
DX*($s=1$)	70	1,073,743,872	(30, 11)
DX*($s=1$)	234	8,396,800	(23, 13)
DX*($s=1$)	287	8,389,120	(23, 9)
DX*($s=1$)	447	4,194,560	(22, 8)
DX*($s=1$)	499	537,001,984	(29, 17)
DX*($s=2$)	122	67,108,992	(26, 7)
DX*($s=2$)	555	262,160	(18, 4)
DX*($s=2$)	608	16,781,312	(24, 12)
DX*($s=3$)	166	67,108,896	(26, 5)
DX*($s=3$)	779	16,809,984	(24, 15)
DX*($s=4$)	323	1,082,130,432	(30, 23)

$0 \le r < w$, $19 \le w \le 30$ are some positive integers. Because the searching space for such B is rather limited, we search over various t for maximum-period DX, DL, and DS generators. Specifically, we start from the default value of t and move upward until a number of Bs are found. We list these generators in Table 2.

5.3. DX* Generators with $p=2^{31}-1$ and $B=2^r\pm2^w$

We search for $B=2^r\pm2^w$ in the class of DX* generators given in (14)–(17), which are slight but significant modifications to DX generators. The results are listed in Table 3.

5.4. Empirical Evaluations

We evaluate the generators listed in Tables 1–3 with the stringent empirical tests in the Crush battery of TestU01 test suite. Each generator is tested with five different starting seeds. No p-values are outside the range of $[10^{-8}, 1-10^{-8}]$ for any of these generators. Note that L'Ecuyer and Simard (2007) considered the test with p-value outside the range of $[10^{-10}, 1-10^{-10}]$ as a "clear failure" with one single initial seed. In other words, with five sets of initial seeds and a larger cutoff point of 10^{-8}, our generators have gone through a tougher examination and passed.

6. Comparison

When evaluating the goodness of various generators, many factors should be considered, including the generating speed, high-dimensional equidistribution property, period length, portability, hardware/software requirement, theoretical justifications, and empirical performances. Most of these factors have been addressed earlier; we discuss the rest in the following subsections.

6.1. Timing Comparison with MRG32k3a

In this subsection, we compare our generators with the popular MRG32k3a generator. As to the generating efficiency, each component MRG of MRG32k3a has two nonzero and nonequal multipliers. In total, MRG32k3a requires four multiplication operations. In contrast, DX generators require no multiplication for $B=2^r\pm2^w$ and only one multiplication for general Bs. Also, according to Table 1 of a well-known paper by L'Ecuyer and Simard (2007), a DX generator (with a general B) is about 50% faster than the MRG32k3a generator. The authors reported that, using a 64-bit computer with an AMD Athlon 64 processor of clock speed 2.4 GHz to generate 10^8 random numbers, two DX generators tested need about 1.4 seconds, whereas the MRG32k3a generator needs 2.0 seconds. Because the generating efficiency of the DX generators does not depend on the order k, the generators in Table 1 are at least as efficient as the DX generators tested by L'Ecuyer and Simard (2007). Moreover, the generating

time can be further reduced for the generators with powers-of-two multipliers, according to the empirical study in Deng et al. (2008a, Table 5). We have performed additional timing evaluations as described below (thanks to the referees for this suggestion).

We time the actual generating time of some DX generators listed in this paper and MRG32k3a using the computer clusters at the High Performance Computing Center, University of Memphis; each computer has a 3.2 GHz Intel Xeon 32-bit processor running in Linux. For MRG32k3a, we use the built-in function `ulec_CreateMRG32k3a` in the TestU01 package to generate 10^8 random variates. Using the built-in function `unif01_TimerSumGenWr`, the total time is 6.81 seconds. As reported in L'Ecuyer and Simard (2007), the generating time is highly dependent on the type of CPU (32-bit versus 64-bit) used, although their relative efficiency remains about the same. We test the timing under the same setting on all DX generators listed in Table 2 with a powers-of-two multiplier B. With the modulus $2^{31}-1$, we can use a more efficient implementation as discussed in Wu (1997) and L'Ecuyer and Simard (1999) by replacing the expensive multiplication and modulo operations with more efficient logical operations. For the purpose of comparison, we also implement the same DX generators using a general-purpose program in which the usual multiplication and modulo operations are used. The timing comparisons are tabulated in Table 4, in which (a) is the time (in seconds) needed to generate 10^8 variates for the *special* implementation with logical operations, and (b) is the time needed for the *general* implementation.

From Table 4, we can see that the timing is not greatly affected when we increase (i) the order k from 7,499 to 20,897 or (ii) the size of s, the number of nonzero terms in the DX generators. In addition, it is found that the special implementation (a) can be two to three times more efficient than its general implementation counterpart (b). Using the generating time, 6.81 seconds, for MRG32k3a as the baseline, the relative efficiency of the special implementation (a) and general implementation (b) is about 4- to 5-fold and 1.5-fold more efficient than MRG32k3a, respectively. We remark here that the cost of the modulo operation is usually (at least for the computing platform used here) much higher than the multiplication operation. Therefore, with $p = 2^{31}-1$, we can save significant generating time by replacing the modulo operation with a much more efficient logical operation for all the generators listed in Tables 1–3. On 64-bit computers (which are now commonplace), the efficiency ratios between MRG32k3a and the DX generators will be smaller than reported in Table 4. (Thanks to the referee who pointed this out.) Also, the generator MRG31k3p proposed in L'Ecuyer and Touzin (2000), which uses two MRGs of order 3 with multipliers of the same form (i.e., sum of two powers of 2), is faster than MRG32k3a.

Table 4 **Time Comparisons Between DX-k-s Generators and MRG32k3a**

	$s=1$		$s=2$		$s=3$		$s=4$	
k	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
	Time needed to generate 10^8 variates (in s)							
7,499	1.23	3.73	1.27	4.28	1.28	4.14	1.38	3.87
20,897	1.23	4.12	1.27	3.86	1.57	4.00	1.44	4.14
	Relative timing of DX-k-s vs. MRG32k3a (6.81 s)							
7,499	0.18	0.55	0.19	0.63	0.19	0.61	0.20	0.57
20,897	0.18	0.61	0.19	0.57	0.23	0.59	0.21	0.61

Note. The letter (a) denotes special implementation, and (b) denotes general implementation.

Any timing comparison of the generating speed is highly hardware- and software-dependent. One may observe different results with various computing platforms (32- or 64-bit), compilers, program implementations, or even operating systems. Compared with some popular generators such as MRG32k3a, the generators provided in this paper are clearly more efficient by either simple counting of the required expensive multiplication operations or actual timing. For some applications, saving one or two seconds for the generation of 10^8 random variates is not important at all and need not be a major factor in selecting good random number generators. However (as pointed out by a referee), for certain simulation applications such as in computer graphics and in particle physics, the speed of generating the random numbers is really the bottleneck, and the simulation programs may run for hours or days.

6.2. Hardware Needed

An MRG of order k requires a memory space of size k to store the state vector. Thus, an MRG of order $k = 20{,}897$ needs 20,897 32-bit integers of memory space, which seems quite memory consuming. However, the cost of computer memory is drastically reduced in this modern computing age, and a memory size as such is now considered as a tiny "requirement" with no additional cost because it is not uncommon to acquire a PC with a memory space of 4 GB or more. If the size of memory to store the current state is indeed a concern, one can certainly consider a smaller k such as the popular MRG32k3a or smaller-order DX generators given in Deng and Xu (2003) and Deng (2005).

6.3. Theoretical Tests for the Generators

Ideally, any subsample (or subsequence) of a useful generator must appear to follow a uniform distribution, and the numbers should appear to be independent of each other. The number of possible successive

t-tuples of the output sequence produced by a generator with p possible values is p^t. An ideal generator would be able to produce all p^t t-tuples with equal frequency for any values of t. As mentioned earlier, an MRG of order k with the maximum period almost achieves this requirement with $t \le k$ because it has the equidistribution property over dimensions up to k. However, its lattice structure can be bad when considering (1) successive t-tuples for $t > k$ or (2) selective (possibly nonsuccessive) tuples of generated variates indexed by the set—say, I—corresponding to the indices i for which $\alpha_{k-i} \neq 0$ in the MRG, because in these cases the considered tuples will lie on the family of equidistant parallel hyperplanes of a higher dimension. See L'Ecuyer (1997) or L'Ecuyer and Touzin (2004).

It is common to perform a theoretical test to search for the best generator among a class of LCGs or small-order MRGs based on their lattice structures in a specific dimension t; for example, the popular spectral test is to calculate the maximum distance $d_t(k)$ between adjacent parallel hyperplanes for a chosen dimension t. A measure suggested by Fishman and Moore (1986) for comparing generators with different values of modulus is the spectral value defined as $S_t(k) \equiv d_t^*(k)/d_t(k)$, where $d_t^*(k)$ is the theoretical minimum of $d_t(k)$; however, the exact values of $d_t^*(k)$ are known only for small t—say, $t \le 8$. For MRGs of any orders, the commonly used performance measures are the maximum distance $d_t(k)$ and the approximated spectral value $p^{-k/t}/d_t(k)$. Because comparing generators based on the spectral value may yield conflicting results in different dimensions t, another figure of merit, M_T, was defined in the literature as the smallest spectral value across some dimensions up to dimension T; for example, $M_T \equiv \min_{k<t<T} S_t(k)$.

All of these figures of merit and the equidistribution property are concerned with the "uniform coverage" of the t-tuples from the generated sequence. For the same dimension $t \le k$, the equidistribution property is a stronger property than these figures of merit in the sense that the former can guarantee not only the existence but also the equal occurring frequency of all possible p^t lattice values of t-tuples, except for the all-zero t-tuple, which is one time less. When $t > k$, the property of equidistribution no longer holds even for the maximum-period MRGs of order k; then it is plausible to use the figures of merit to further differentiate these maximum-period MRGs. Sezgin (2006) gave a long list of references for many other measures to assess the lattice structure. However, because of the increasing computational complexity, t cannot be much larger than k, especially when k is large. Because the order k of the generators proposed in this paper is already very large (i.e., 7,499 or 20,897), there is only a minor "marginal effect" to have better figures of merit.

If uniform coverage of the generated sequence in a high-dimensional space is a major concern, then all of our proposed MRGs of order 20,897 are, so far, the best because they all have the dimensions of equidistribution much larger than that of current popular generators. Under the equidistribution criterion, a full-period MRG of a large order is better than a much smaller-order MRG. Therefore, any MRG of order 20,897 listed in this paper outperforms (in terms of the lattice structure for successive tuples) the "best" MRG of order 7,499 (if it can be found). For the same reason, many popular small-order generators (e.g., the popular MRG32k3a) are doomed to be inferior to these large-order MRGs from this aspect. Nevertheless, MRG32k3a is a good combined generator in which each of the two component MRGs has a property of equidistribution up to three dimensions. L'Ecuyer (1999) showed that MRG32k3a is "well behaved" in all dimensions up to at least 45 with the figure of merit, M_{45} (approximately 0.6225). Having the equidistribution property up to k dimensions, all MRGs of order $k = 20{,}897$ or $k = 7{,}499$ proposed in this paper are "better" than MRG32k3a under any of these uniform coverage criteria. L'Ecuyer and Simard (2007) also reported that both MRG32k3a and some large-order MRGs (including DX-1597 and DX-47, proposed in Deng 2005) passed the stringent Crush battery of tests in the TestU01 library. In addition, they reported that all LCGs, including the "best" (under spectral tests) LCGs found in Fishman and Moore (1986), failed the Crush battery badly.

To further compare all maximum-period MRGs of the same order k (i.e., they all have the nice equidistribution property for dimensions up to k), we need to consider dimensions where t is larger than k (or some specifically selected dimensions) to see any difference in terms of "better lattice structure." For *general* large-order MRGs with no special forms, it is likely to be computationally infeasible to apply the exact standard spectral test algorithm, which takes exponential time in the dimension t, to compute the maximum distance, $d_t(k)$, between parallel hyperplanes in dimension t higher than k, especially for generators of order k as large as 7,499 or 20,897. Nevertheless, L'Ecuyer (1997) gave a simple lower bound, $(1 + \sum_{i=1}^{k} \alpha_i^2)^{-1/2}$, for $d_t(k)$ and concluded that a "good" MRG should have a large sum of squares of coefficients, $\sum_{i=1}^{k} \alpha_i^2$ (but not vice versa). This simple lower bound can be used to "detect" generators with potential bad lattice structures, but, as a lower bound, the information is not sufficient to compare the "actual" lattice structures among generators. Unless this simple lower bound can approximate equally well the actual value

for various MRGs, one cannot mathematically compare the sizes of the actual values based only on their lower bounds. Even if such comparisons can be made, the differences and their practical effect on the performances of random number generators may be only marginal because of the high-dimensional equidistribution property. For example, according to our empirical study using various sizes of multiplier B for DX generators with a large order k, we see no practical differences. As stated in §5, they all pass the stringent tests in the Crush battery of TestU01.

Because it is time consuming to find large-order maximum-period MRGs (with the same efficiency), it is very hard to find "optimal" parameters for generators in some particular dimensions larger than k based on figures of merit. Here, we provide a general guideline to avoid a bad lattice structure in a high dimension. First, among DX-k-s generators, we prefer the ones with a larger value of multiplier B and/or a larger s to have more and larger nonzero terms. On the other hand, larger values of s or larger values of B tend to make the generator less efficient and less portable. Second, for DL generators in (10) and DS generators in (12), they shall have a small lower bound for $d_t(k)$ in $t=k+1$ dimensions because they have many nonzero terms. However, from their $(k+1)$st recurrence equations in (11) and (13), respectively, we can see that they are likely to have a bad lattice structure in $t=k+2$ dimensions, especially for DL generators. We remark that, if a generator with a better lattice structure in high dimensions is indeed needed, we recommend using any of the efficient MRGs (other than $\min B$) of order $k=20{,}897$ listed in Tables 1–3. As mentioned in §5, MRGs with $\min B$ listed in those tables can be useful for other purposes.

To sum up, all of our proposed MRGs of order 20,897 have a nice property of equidistribution in all dimensions up to 20,897 with some possible bad lattice structures (especially for MRGs with a small multiplier and few nonzero terms) in dimensions larger than 20,897. However, there is no need to rule out a generator of a large order k simply because it has a bad lattice structure in $k+1$ or $k+2$ dimensions. After all, no generator can claim to be a perfect random number generator; for most generators, there is always a trade-off between better generating efficiency (by considering fewer nonzero terms) and other criteria such as better lattice structure in a higher dimension.

7. Concluding Remarks

We have found many efficient maximum-period MRGs of a large order k with $k=7{,}499$ and $k=20{,}897$ for the prime modulus $p=2^{31}-1$. The successful search was achieved by utilizing some results in number theory as well as some powerful factorization algorithms. The large order of maximum-period MRGs automatically gives the nice property of high-dimensional equidistribution. We conducted an empirical timing study and showed that, with efficient implementation, the proposed generators can be more than five times as efficient as the popular MRG32k3a. Two key factors for the great generating efficiency are the choices of (i) the particular modulus $p=2^{31}-1$ and (ii) the specific form of the multiplier $B=2^r\pm 2^w$. In addition, all the generators reported in this paper have extremely long periods and great empirical performances.

With great generating efficiency and aforementioned nice properties, these MRGs would be useful for many scientific applications, especially for those applications with very large-scale simulations. Currently, we are exploring various possibilities of transforming large-order MRGs into nonlinear generators to enhance their security properties, an essential requirement for cryptography applications, while maintaining most of the nice properties stated above.

Acknowledgments

This research was partially supported by the National Science Council of Taiwan, Republic of China, Grant, NSC98-2118-M-009-004-MY3 and NSC97-2118-M-009-002-MY2, National Center for Theoretical Sciences, Center of Mathematical Modeling and Scientific Computing at National Chiao Tung University. This work was done while the first author was visiting the Institute of Statistics, National Chiao Tung University, Hsinchu, Taiwan. The authors also acknowledge the use of the high-speed computing facility provided by the University of Memphis for this research. The authors are grateful for the comments and suggestions by the area editor and two anonymous referees who made a significant contribution to the improvement of this paper.

References

Alanen, J. D., D. E. Knuth. 1964. Tables of finite fields. *Sankhyā Ser. A* **26**(4) 305–328.

Crandall, R., C. Pomerance. 2000. *Prime Numbers: A Computational Perspective*. Springer-Verlag, New York.

Damgård, I., P. Landrock, C. Pomerance. 1993. Average case error estimates for the strong probable prime test. *Math. Comput.* **61**(203) 177–194.

Deng, L.-Y. 2004. Generalized Mersenne prime number and its application to random number generation. H. Niederreiter, ed. *Monte Carlo and Quasi-Monte Carlo Methods 2002*. Springer-Verlag, Berlin, 167–180.

Deng, L.-Y. 2005. Efficient and portable multiple recursive generators of large order. *ACM Trans. Modeling Comput. Simulation* **15**(1) 1–13.

Deng, L.-Y. 2008. Issues on computer search for large order multiple recursive generators. S. Heinrich, A. Keller, H. Niederreiter, eds. *Monte Carlo and Quasi-Monte Carlo Methods 2006*. Springer-Verlag, Berlin, 251–261.

Deng, L.-Y., H. Xu. 2003. A system of high-dimensional, efficient, long-cycle and portable uniform random number generators. *ACM Trans. Modeling Comput. Simulation* **13**(4) 299–309.

Deng, L.-Y., H. Li, J.-J. H. Shiau. 2009. Scalable parallel multiple recursive generators of large order. *Parallel Comput.* **35**(1) 29–37.

Deng, L.-Y., R. Guo, D. K. J. Lin, F. Bai. 2008a. Improving random number generators in the Monte Carlo simulations via twisting and combining. *Comput. Phys. Comm.* **178**(6) 401–408.

Deng, L.-Y., H. Li, J.-J. H. Shiau, G. H. Tsai. 2008b. Design and implementation of efficient and portable multiple recursive generators with few zero coefficients. S. Heinrich, A. Keller, H. Niederreiter, eds. *Monte Carlo and Quasi-Monte Carlo Methods 2006*. Springer-Verlag, Berlin, 263–273.

Fishman, G. A., L. R. Moore III. 1986. An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$. *SIAM J. Sci. Stat. Comput.* **7**(1) 24–45.

Knuth, D. E. 1998. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed. Addison-Wesley, Reading, MA.

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

L'Ecuyer, P. 1997. Bad lattice structures for vectors of nonsuccessive values produced by some linear recurrences. *INFORMS J. Comput.* **9**(1) 57–60.

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

L'Ecuyer, P., R. Simard. 1999. Beware of linear congruential generators with multipliers of the form $a = \pm 2^q \pm 2^r$. *ACM Trans. Math. Software* **25**(3) 367–374.

L'Ecuyer, P., R. Simard. 2007. TestU01: A C library for empirical testing of random number generators. *ACM Trans. Math. Software* **33**(4) Article 22.

L'Ecuyer, P., R. Touzin. 2000. Fast combined multiple recursive generators with multipliers of the form $a = \pm 2^q \pm 2^r$. J. A. Joines, R. R. Barton, K. Kang, P. A. Fishwick, eds. *Proc. 2000 Winter Simulation Conf.*, IEEE Press, Pistacaway, NJ, 683–689.

L'Ecuyer, P., R. Touzin. 2004. On the Deng-Lin random number generators and related methods. *Statist. Comput.* **14**(1) 5–9.

L'Ecuyer, P., F. Blouin, R. Couture. 1993. A search for good multiple recursive random number generators. *ACM Trans. Modeling Comput. Simulation* **3**(2) 87–98.

Lenstra, H. W., Jr. 1987. Factoring integers with elliptic curves. *Ann. Math.* **126**(2) 649–673.

Lidl, R., H. Niederreiter. 1994. *Introduction to Finite Fields and Their Applications*, revised ed. Cambridge University Press, Cambridge, MA.

Marse, K., S. D. Roberts. 1983. Implementing a portable FORTRAN uniform $(0, 1)$ generator. *Simulation* **41**(4) 135–139.

Pollard, J. M. 1974. Theorems on factorization and primality testing. *Proc. Cambridge Philos. Soc.* **76**(3) 521–528.

Pollard, J. M. 1975. A Monte Carlo method for factorization. *BIT* **15**(3) 331–334.

Pollard, J. M. 1993. Factoring with cubic inetegers. A. K. Lenstra, H. W. Lenstra Jr., eds. *The Development of Number Field Sieve.* Lecture Notes in Mathematics, Vol. 1554. Spring-Verlag, New York, 4–10.

Riesel, H. 1994. *Prime Numbers and Computer Methods for Factorization*, 2nd ed. Birkhäuser, Boston.

Sezgin, F. 2006. Distribution of lattice points. *Computing* **78**(2) 173–193.

Wu, P.-C. 1997. Multiplicative, congruential random-number generators with multiplier $\pm 2^{k1} \pm 2^{k2}$ and modulus $2^p - 1$. *ACM Trans. Math. Software* **23**(2) 255–265.