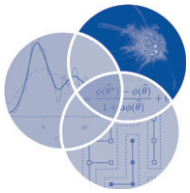
Advanced Review



Developments in pseudo-random number generators

Lih-Yuan Deng* and Dale Bowman

Monte Carlo simulations have become a common practice to evaluate a proposed statistical procedure, particularly when it is analytically intractable. Validity of any simulation study relies heavily on the goodness of random variate generators for some specified distributions, which in turn is based on the successful generation of independent variates from the uniform distribution. However, a typical computer-generated pseudo-random number generator (PRNG) is a deterministic algorithm and we know that no PRNG is capable of generating a truly random uniform sequence. Since the foundation of a simulation study is built on the PRNG used, it is extremely important to design a good PRNG. We review some recent developments on PRNGs with nice properties such as high-dimensional equi-distribution, efficiency, long period length, portability, and efficient parallel implementations.

How to cite this article:
WIREs Comput Stat 2017, 9:e1404. doi: 10.1002/wics.1404

Keywords: statistical computing, numerical analysis, Monte Carlo methods, random number generation

INTRODUCTION

In most fields of scientific research today, it is common to use computer simulation to evaluate statistical methodologies, to take random samples from target distributions, and to use Markov Chain Monte Carlo (MCMC) for simulating posterior distributions. The National Air Force Research Laboratories report that a large portion of supercomputer resources are currently spent on Monte Carlo computations. The results of a computer-intensive study based on random sampling depend largely on the quality of (pseudo)-random numbers available, which in turn depends on the properties of the pseudo-random number generator (PRNG) used. Typically, to produce a sequence of random variables from a specified distribution, first independent random integers from 0 to an integer m are generated and then transformed into variates in [0, 1] by dividing by m. These uniform variates are transformed again to variables from the desired distribution. Several commonly used PRNGs are presented in this article.

There are typically five major issues considered when selecting or comparing PRNGs. These are (1) uniformity and the equi-distribution property (2) period length (3) computing efficiency and portability (4) statistical justifications and empirical performance and (5) ability to parallelize and generate substreams. The equi-distribution property up to k dimensions ensures that virtually every d-tuple ($d \leq k$) of integers appears exactly the same number of times over the entire period of the generator. The properties of equi-distribution over high dimension spaces and longer period length can be achieved by selecting a class of generators that have these properties. Computing efficiency and portability are then achieved by carefully chosen designs of these generators. The empirical properties and statistical justifications can inspire the development of ever better PRNGs with stronger empirical performances. The issues of parallelization are becoming increasingly more important with the increasing use and access to ever more powerful computing resources. The use of pseudo random number generators on single

*Correspondence to: lihdeng@memphis.edu

Department of Mathematical Sciences, University of Memphis, Memphis, TN, USA

Conflict of interest: The authors have declared no conflicts of interest for the article.

processors has been well studied, but less work has been done in the case of parallel computing. Parallel computing is a common way to speed up computations by breaking a task into smaller relatively independent subtasks that can be handled by individual processors in parallel. In this case, in order to obtain good overall results, it is necessary that good parallel random number generators be used. The design and implementation of suitable and independent parallel PRNGs are discussed in this article.

This article is organized into two main sections—the first reviews classical PRNGs, extensions, and improvements of these generators and the second discusses issues relating to parallelization. The first section begins with a review of the classical linear congruential generator (LCG) and the use of the technique of combination to improve LCGs. Linear feedback shift registers (LFSRs) are another class of generators discussed next. Important extensions of LCGs, namely multiple recursive generators (MRGs) and matrix congruential generators (MCGs), are next and some theoretical properties and results are discussed. Large order MRGs have advantages of long period length, and equi-distribution in high dimensions as well as better empirical performances. A brief discussion is given on procedures that can be used to search for MRGs of high order that will retain maximal properties of period length and efficiency. A class of portable and efficient MRGs is considered next and finally a review of some generators that can be found in R programming language is given.

Efficiency in computations can be greatly increased if multiple processors can be assigned to perform multiple independent simulations. Thus, there is a need for parallel PRNGs that can efficiently produce high quality streams of independent random numbers. The second part of the article examines parallel random number generators using LCGs and MRGs as baseline generators. Traditional methods for parallel PRNGs use a single baseline generator and choose seeds that are far apart to assign different streams to different processors. A newer proposed method uses results from number theory to automatically produce different PRNGs each with maximal period, efficiency, and great empirical performances to assign to various processors. A comparison is made between the methods. The article ends with some conclusions about the developments reviewed.

Some Classical Random Number Generators

We consider a general form of a PRNG as

$$X_i = f(X_{i-1}, \cdots, X_{i-k}) \bmod p, \quad i \geq 0, \tag{1}$$

where f is a function of the most recent k integers in the past and X_{-k}, $X_{-(k-1)}$, $\cdots$, X_{-1} are initial seeds taken from $\mathbb{Z}_p = \{0,1, \cdots, p-1\}$. Most PRNGs are of this general form with some appropriate choice of function f. We discuss some of the popular choices for f shortly.

For many computer applications, the goal of PRNGs is to produce a sequence of variates that is very hard to distinguish from a sequence of (truly) random numbers. Therefore, an ideal PRNG should satisfy the property of high-dimensional equi-distribution, efficiency, long period length, and portability, also known as the HELP property in Deng.[1]

Linear Congruential Generators

If f is a linear function with $k = 1$, the generator (Eq. (1)) is an LCG, first proposed by Lehmer.[2] Its sequence is obtained by

$$X_i = (BX_{i-1} + A) \bmod p, \quad i \geq 0, \tag{2}$$

where X_i, A, B, and p are nonnegative integers and $X_{-1} \neq 0$ is chosen from $\mathbb{Z}_p$ as a seed. If A is not zero, it is possible to achieve the full period p. However, according to Marsaglia,[3] the 'effective period' cannot be greater than the period of the corresponding LCG with $A = 0$ as

$$X_i = BX_{i-1} \bmod p, \quad i \geq 0, \tag{3}$$

where $X_{-1} \neq 0$. When p is a prime number and B is a primitive element (also known as primitive root) modulo p, the LCG in Eq. (3) has period of $p-1$. B is a primitive root modulo p, if for any prime factor q of $(p-1)$, $B^{(p-1)/q} \neq 1 \bmod p$. Furthermore, if B is a primitive root, then $B^r \bmod p$ is also a primitive root as long as r and $(p-1)$ are relatively prime, $\gcd(r, p-1) = 1$. In fact, we can produce all primitive roots as $B^r \bmod p$ with $\gcd(r, p-1) = 1$. Hence, the total number of primitive roots is $\phi(p-1)$, where $\phi(x)$ is the Euler's totient function counting the number of integers between 1 and x that are relatively prime to x. Deng et al.[4] used this property to develop a procedure to automatically produce many maximum-period LCGs for parallel simulation to be discussed later.

For the case of $p = 2^{31}-1$, 7 is the smallest primitive element; but $B = 7$ is not a suitable multiplier for LCG because it is too small.[5] A minimum standard was proposed by Park and Miller[6] by choosing an LCG with $p = 2^{31}-1$ and $B = 7^5$, where

5 is chosen because it is the smallest exponent $r > 1$ with $\gcd(r, p-1) = 1$. This LCG has been used in most computer systems and packages and has period of $p-1 \approx 2.1 \times 10^9$. To further speed up the generating efficiency, one can consider a special form of the multiplier $B = 2^r \pm 2^w$ for LCGs with $p = 2^{31}-1$. A multiplier of this form, called powers-of-two decomposition, was first suggested by Wu[7] for LCGs. It can result in faster computation because one can replace more expensive multiplication and modulus operations with some more efficient logical shift and addition operations. Multipliers with powers-of-two decomposition can be useful to increase the generating efficiency for other PRNGs to be discussed later.

Clearly, there is a simple linear relationship between two successive variates produced by an LCG in Eq. (3). Marsaglia[8] was the first to show that the successive overlapping sequences of d random numbers fall on at most $(d!p)^{1/d}$ planes. Therefore, even when d is of moderate size, successive d-tuples in the output sequence by LCG will lie in a simple lattice structure.

LCGs were popular for their simplicity, efficiency, and well-known theoretical properties. However, LCGs are not recommended because they have relative short periods by today's standards and they lack equi-distribution in dimensions higher than 1. Furthermore, LCGs have questionable empirical performances and all LCGs have failed badly some stringent empirical tests in L'Ecuyer and Simard.[9]

There are several proposed methods to improve the performance of LCGs. For example, Marsaglia[10] proposed the 'multiply-with-carry' (MWC) generator given by

$$X_i = (BX_{i-1} + A_{i-1}) \mod m, \qquad (4)$$

and compute $A_i = \lfloor \frac{(BX_{i-1} + A_{i-1})}{m} \rfloor$ to be used as the 'carry' for the next iteration and the initial A_{-1} can be set by the user. Clearly, this class of generators is similar in form to Eq. (2) with 'carry,' A_i, as a 'feedback' for the next iteration. MWC was originally proposed as a generalization of the 'add-with-carry' generator of Marsaglia and Zaman.[11] For efficiency considerations, it is recommended that $m = 2^{32}$. Additional theoretical studies on the MWCs can be found in Couture and L'Ecuyer[12] and Goresky and Klapper.[13] Compared to LCGs, MWC generators have much longer periods, better distribution properties, and they are reasonably efficient to compute. We consider other methods to improve LCGs next.

Combination Generators to Improve LCGs

The performance of some generators can be improved by combining two or more pseudo-random sequences. Wichmann and Hill[14] proposed the first combined generator that produced sequences by adding random numbers generated by three simple LCGs and then taking the fractional part as the random number. Specifically, three simple LCGs considered by Wichmann and Hill[14] are:

1. $X_i = 171X_{i-1} \mod 30269$ (m_1)
2. $Y_i = 172Y_{i-1} \mod 30307$ (m_2)
3. $Z_i = 170Z_{i-1} \mod 30323$ (m_3)

The combination generator is

$$U_i = X_i/m_1 + Y_i/m_2 + Z_i/m_3 \mod 1 \qquad (5)$$

and its period length is increased to $\mathrm{LCM}(m_1-1, m_2-1, m_3-1) \approx 6.95 \times 10^{12}$. Zeisel[15] showed that the combination generator is equivalent to another LCG with a large multiplier ($B = 16555425264690$) and a large modulus ($p = 27817185604309 = m_1 \times m_2 \times m_3$). This equivalent generator is found using the well-known Chinese Remainder Theorem to solve

$$B = 171 \mod m_1, \quad B = 172 \mod m_2, \quad B = 170 \mod m_3.$$

The formula to find B is

$$B = (B_1 + B_2 + B_3) \mod (m_1 m_2 m_3),$$

where B_1, B_2, B_3 can be found by taking the multiplicative inverses as

$$B_1 = 171(m_2 m_3)\left((m_2 m_3)^{-1} \mod m_1\right),$$

$$B_2 = 172(m_1 m_3)\left((m_1 m_3)^{-1} \mod m_2\right),$$

$$B_3 = 170(m_1 m_2)\left((m_1 m_2)^{-1} \mod m_3\right).$$

Statistical Justification and Intuitive Explanations

Wichmann and Hill[14] did not provide any theoretical justification for their combination generator. Several theoretical justifications of combined generators have been given in the literature. See, for example, Gentle[16] and Brown et al.[17] Specifically, Marsaglia[18] proved that a combined generator yields a distribution closer to or at least no farther from the uniform distribution than that of the corresponding individual

generators. Deng and George[19] provided an additional theoretical justification by showing that a combined generator should improve the uniformity with the following theorem:

Theorem 1. *Let X_i be a continuous random variable on [0, 1] with the probability density function $f_i(x_i)$ such that $|f_i(x_i)-1| \le \varepsilon_i$ for some constant ε_i, $i = 1,2,\cdots, n$ and assume that the X_i's are independent. Let $Y = \sum_{i=1}^{n} X_i \bmod 1$. Then the probability density function of Y, f(y), satisfies*

$$|f(y)-1| \le \prod_{i=1}^{n} \varepsilon_i.$$

Therefore, we can improve the n component (possibly 'defective') generators, represented as X_i's ($i = 1,2,\cdots, n$) whose distributions are close but not exactly uniform, by combining these random variate generators. Note the assumption on the existence of the density of X_i ($i = 1,2,\cdots, n$) is unrealistic. However, we can treat the PRNGs as simply finite-bit realizations of a random variable, X_i with a probability density function. In addition, it may also be unrealistic to assume the variates X_i's generated by these generators are independent of each other. Deng et al.[20] showed that the independence assumption in Theorem 1 can be further relaxed. In particular, they considered an extreme case in which the X_i's are all identical to a random variable X. That is, $Y = nX \bmod 1$. They showed that the asymptotic distribution of Y is a $U(0, 1)$ distribution as given in the following theorem:

Theorem 2. *Let X be a random variable on [0, 1] with probability density function $f_X(x)$. Let $Y_n = nX \bmod 1$. Then the probability density function of Y_n, $f_n(y)$, satisfies*

$$|f_n(y)-1| \to 0, \quad \text{as } n \to \infty.$$

According to the above theorems, one can generate a 'flatter' distribution by either adding up several variables or multiplying a random variate with a large constant and then taking its fractional part.

To provide some formal justification for this result, Deng et al.[20] and Deng[21] gave a simple intuitive explanation and we provide some additional graphical illustrations below. It is intuitive to see that the probability density function, say $f(y)$, for random variable $Y = \sum_{i=1}^{n} X_i \bmod 1$, will become 'flatter' as long as the variance of Y becomes larger, regardless of whether the individual components, X_i, are independent or not. Hence, the probability density function $f(y)$, will become very flat and the 'mod 1 operation' will accumulate the probability density functions over the unit interval making the resulting random variable very close to $U(0, 1)$. Even under the situation that all the X_i are duplicates of each other ($X_i = X$), it is straightforward to see $Y = nX \bmod 1$ will converge to a $U(0, 1)$ distribution. One can illustrate this from the plots of the probability density function's for X shown in red, $2X$ shown in green and $5X$ shown in blue with $X \sim \text{Gamma}(2, 1)$, in Figure 1(a), which shows that the probability density function for $Y = cX$ becomes flatter as c increases. Next, we plot the probability density functions for ($X \bmod 1$) shown in red, ($2X \bmod 1$) shown in green, and ($5X \bmod 1$) shown in blue with $X \sim \text{Gamma}(2, 1)$ as in Figure 1(b). Thus, it is easy to confirm the above argument even without a formal proof.

It is interesting to note that our previous intuitive argument can be extended to k-dimensions. That is, it can be easily extended to the combination of n k-dimensional random vectors with $Y = X_1 + X_2 + \cdots + X_n \bmod 1$ and we can show a similar result for $Y = n\,X \bmod 1$, where X is a k-dimensional random vector. Almost exactly the same argument can be used by stretching (in each of the k dimensions) the joint probability density function which will flatten the 'surface' of the joint probability density function. Taking the 'mod 1 operation' on the random vector will cause it to converge to a uniform distribution of $U[0, 1]^k$. Theoretical justifications for combining random vectors can be found in Deng and Chu,[22] Deng et al.,[20] and Deng.[21]

One can also illustrate the above argument using a simple example with $(X, Y) \bmod 1$, where (X, Y) has a bivariate normal distribution with a large correlation coefficient $\rho = 0.99$. It is clear, from Figures 2 and 3, that even after taking 'mod 1' operation, the probability density function is not very close to a uniform distribution over $[0, 1]^2$. However, the successive plots of probability density functions for $(2X, 2Y) \bmod 1$, $(4X, 4Y) \bmod 1$, $(8X, 8Y) \bmod 1$, show the probability density function is indeed approaching a uniform distribution over $[0, 1]^2$.

We remark here that the previous theory is based on the assumption that the variable X is a continuous random variable with a probability density function. In practice, a random number generator can only produce a finite number of discrete points in [0, 1], and hence the generated variates are only approximations to realizations of the corresponding continuous random variables. However,

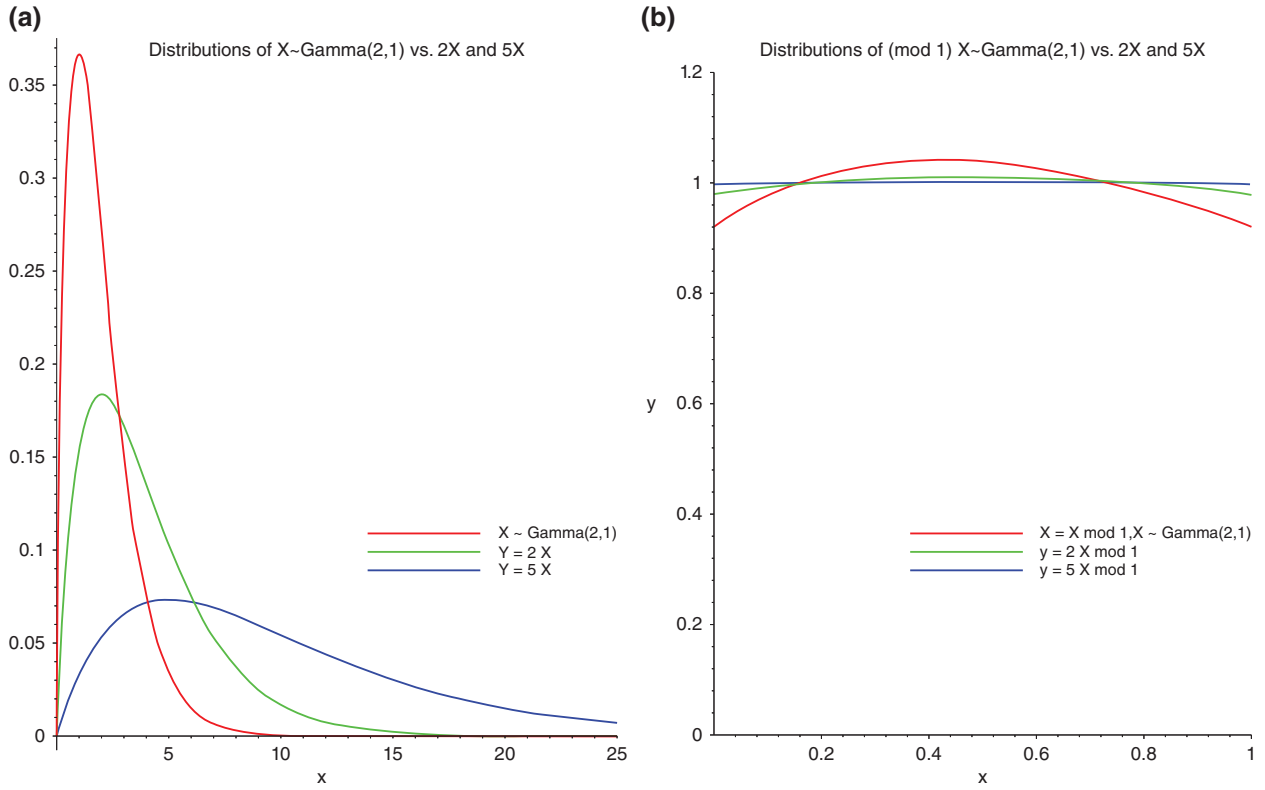


FIGURE 1 | Plots of probability density functions. Panel (a) is for X (red), $2X$ (green), and $5X$ (blue) with $X \sim$ Gamma(2, 1) and panel (b) is for $Y = cX$ mod 1, $c = 1, 2,5$.

the assumption of the existence of a probability density function can yield some exact and simple expressions to shed light on the theoretical justification. In fact, these assumptions have been implicitly made on the random variates produced by random number generators when using them. For example, the inverse CDF method is based on the assumption that these generated discrete points in [0, 1] form random samples from a continuous uniform distribution. Generally speaking, we need component generators to be able to produce points that are 'dense' enough over the region [0, 1].

One can apply the previous argument to consider another method of combining generators, twisting and combining (TAC), to improve the performance of PRNGs by improving the uniformity of the variates.[23] In a general sense, MWC as mentioned in Eq. (4) can also be viewed as a combination generator between the X_i and its 'carry' A_i. While the distributions of both component generators (X_i and A_i) are not close to the desired uniform distribution, the combination generator (MWC in Eq. (4)) should have a distribution closer to the uniform distribution.

While the idea of combining n generators can be helpful to improve the generators, it may significantly slow down the generating efficiency when n is large. We need to consider other classes of efficient generators with extremely long period length and some great distributional properties.

Linear Feedback Shift Register

A special class of general PRNGs with modulus, $p = 2$ and f in Eq. (1) a linear function is the LFSR. The coefficients (α_j) for the LFSRs are either $\alpha_j = 0$ or 1 for $1 \le j \le k-1$ with $\alpha_k = 1$ and an initial nonzero binary vector, $(X_{-k}, \ldots, X_{-1})$. The LFSR will achieve the maximum period 2^k-1 if the necessary and sufficient condition that the polynomial

$$f(x) = x^k - \alpha_1 x^{k-1} - \cdots - \alpha_k \tag{6}$$

is a primitive polynomial over $\mathbb{Z}_2$ is met.[5] A computationally efficient algorithm that produces binary sequences that achieve the maximum period of 2^k-1 is given using a primitive polynomial with only three terms as

$$f(x) = x^k - x^{k-t} - 1, \quad 1 \le t \le k-1. \tag{7}$$

and using a k-th order linear recurrence equation

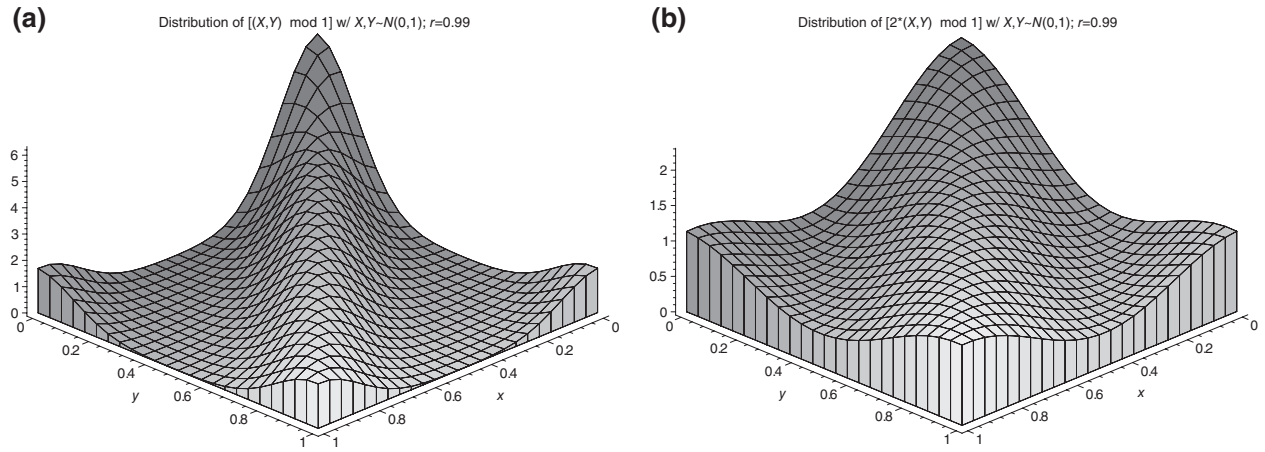


FIGURE 2 | Plots of probability density functions for a bivariate normal vector with $\rho = 0.99$, (X, Y). Panel (a) is for (X, Y) mod 1 and panel (b) is for $(2X, 2Y)$ mod 1.

$$X_i = (X_{i-k} + X_{i-t}) \mod 2. \tag{8}$$

Watson[24] tabulated some primitive polynomials with degree up to 100 and Knuth[5] gives references for more primitive polynomials with modulus 2.

A generalization of the LFSR, called 'generalized feedback shift-register' (GFSR) was proposed by Lewis and Payne[25] to increase generating speed. A special case of the GFSR, an additive lagged Fibonacci generator, is discussed in Marsaglia.[18] The equation for an additive lagged Fibonacci generator is

$$X_i = X_{i-k} \pm X_{i-t} \mod 2^w. \tag{9}$$

The maximum attainable period for these generators is $(2^k-1)2^{w-1}$. For small values of k, lagged Fibonacci generators may have poor statistical properties,[26] while increasing k improves their performance but increases memory requirements.

R software includes 'Knuth-TAOCP' and 'Knuth-TAOCP-2002,' a more recent variant with different initialization, as user-selectable PRNGs. They are lagged Fibonacci generators considered in Knuth[5] as

$$X_i = X_{i-100} - X_{i-37} \mod 2^{30}. \tag{10}$$

MT19937

Matsumoto and Kurita[27] proposed a twisted version of a GFSR. This is accomplished by multiplying a variate in the generator by a suitably chosen matrix. Perhaps the most popular PRNG in current use for applications in computer simulation, the Mersenne

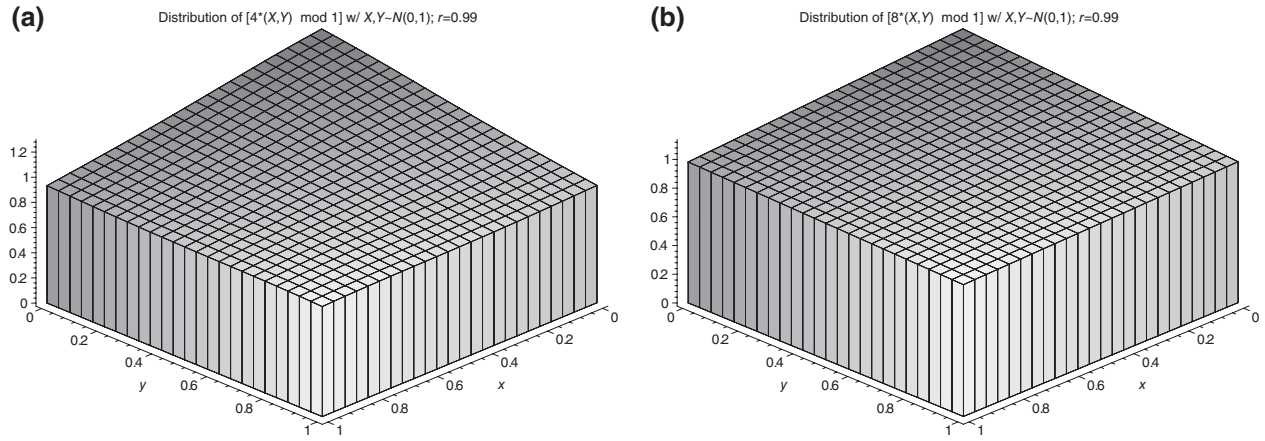


FIGURE 3 | Plots of probability density functions for a bivariate normal vector with $\rho = 0.99$, (X, Y). Panel (a) is for $(4X, 4Y)$ mod 1 and panel (b) is for $(8X, 8Y)$ mod 1.

Twister, proposed by Matsumoto and Nishimura,[28] is an example of a twisted GFSR.[16] The MT19937 generator, based on the Mersenne prime $2^{19937}-1$, is the most widely implemented version of the Mersenne Twister algorithm and is the most widely used PRNG for computer simulations.[29]

The MT19937 uses a matrix linear recurrence (with more than 100 nonzero terms) of order 19937 over the binary finite field, $\mathbb{Z}_2$, and is related to MCGs[16] to be discussed in the next section. MT19937 can provide fast generation of 32-bit, pseudo-random numbers, has a period length of $2^{19937}-1 \approx 10^{6001}$ and equi-distribution up to 623 dimensions. MT19937 is the default generator for many popular software systems including Microsoft Excel, Python, standard C++, and R. It is also one of the PRNGs used in SPSS (SPSS Inc., Chicago) and in SAS (SAS Institute Inc., Cary, NC).

Although MT19937 is currently the most popular PRNG used for its long period length and high-dimensional equi-distributional property, MT19937 consistently fails linear complexity tests in the TestU01 test suite, see L'Ecuyer and Simard.[9]

Two Extensions of LCG: MRG and MCG

LCGs are very efficient. However, by today's standards, LCGs have poor empirical performances, short periods, and inadequate uniformity in dimensions higher than one. These properties and others have earned the LCG a reputation as an unsuitable generator for modern, often large scale, simulation tasks.

To find better classes of PRNGs, it is natural to consider extensions of the LCG. The first is the MRG which generates the next pseudo-random number based on a k-th order equation instead of the first order equation. MRGs indeed have better empirical performances, longer periods, and higher dimensions of uniformity. But they come at two costs: efficiency and finding the right parameters to meet maximum period length, especially as the order k or modulus p increases. Next, we review the basics of MRGs, considerations when searching for MRGs, and special designs to improve efficiency.

The second extension of the LCG is the MCG, which extends the LCG to k-dimensions. Similar to the manner in which MRGs improve on LCGs, MCGs also benefit from better empirical performance, longer periods, and higher dimensions of uniformity. MCGs also share similar costs as the MRGs. Later MCGs and their mathematical properties are briefly discussed.

Extending the LCG with the proper design could prove to be the best balance of better empirical and theoretical performance while not costing too much in terms of efficiency or in searching for parameters that yield maximum period length for a given order k and modulus p.

MRG: The k-th Order Extension of the LCG

The k-th order linear recurrence for an MRG can be defined as

$$X_i = \alpha_1 X_{i-1} + \alpha_2 X_{i-2} + \cdots + \alpha_k X_{i-k} \bmod p, \quad i \geq 0, \tag{11}$$

where $\alpha_1, \alpha_2, \ldots, \alpha_k$ are integers in $\mathbb{Z}_p = \{0,1, \ldots, p-1\}$, $\alpha_k \neq 0$, and we can choose any k not-all-zero values as starting seeds, $(X_{-k}, X_{-(k-1)}, \cdots, X_{-1}) \neq (0, 0, \cdots, 0)$. When the order $k = 1$, the MRG reduces to an LCG. The LFSR can be considered as a special case of MRG with $p = 2$. It is well-known that the maximum period of the MRG is p^k-1 which is much larger than the period length of LCG $(p-1)$. Checking whether an MRG has maximum period length is equivalent to checking whether its characteristic polynomial

$$f(x) = x^k - \alpha_1 x^{k-1} - \alpha_2 x^{k-2} - \cdots - \alpha_k \tag{12}$$

is a k-degree primitive polynomial over $\mathbb{Z}_p$. See, for example, Ref 5. Note that there is a simple relationship between an MRG's companion matrix and its characteristic equation. For an MRG defined in Eq. (11), we can define its corresponding companion matrix

$$\mathbf{M}_f = \begin{pmatrix} 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & 1 \\ \alpha_k & \alpha_{k-1} & \alpha_{k-2} & \ldots & \alpha_1 \end{pmatrix}, \tag{13}$$

and it is straightforward to see that

$$f(x) = \det(x\mathbf{I} - \mathbf{M}_f) \bmod p. \tag{14}$$

For simplicity, we let MRG(k, p) denote the class of maximum-period k-th order MRGs with prime modulus p.

Theoretical Properties of MRGs

It is widely known that MRG(k, p)'s have strong statistical justification and excellent empirical

performance. For more on the statistical justification for MRGs see Deng.[21] Furthermore, MRGs have several other very desirable properties including a huge period length, strong statistical justification, and excellent empirical performance. In addition, these generators have the equi-distribution property up to order k, that is, over its entire, enormous period of p^k-1, every d-tuple $(1 \le d \le k)$ of integers in $\mathbb{Z}_p^d$ appears exactly the same number of times (p^{k-d}), with the exception of the all-zero tuple which appears one time less. See, Lidl and Niederreiter,[30] Theorem 7.43. As the order k increases for a given prime modulus p, the large period and equi-distribution properties become more advantageous. Therefore, when they are available, MRGs with larger order k are preferred. However, finding the parameters $\alpha_1, \alpha_2, \ldots, \alpha_k$ for order k and modulus p such that the MRG is of maximum period is difficult as k or p increases. Furthermore, once they are found, efficiently implementing them requires clever design.

MCG: The k-th Dimensional Extension of the LCG

The MCG is another natural k-dimensional extension of the LCG:

$$\mathbf{X}_i = \mathbf{B}\mathbf{X}_{i-1} \bmod p, \quad i \ge 0, \tag{15}$$

where $\mathbf{X}_i$ is a k-dimensional vector in $\mathbb{Z}_p^k$, $\mathbf{X}_{-1}$ is a nonzero vector, and the multiplier matrix $\mathbf{B}$ is a $k \times k$ matrix in $\mathbb{Z}_p^{k\times k}$. MCG was considered by many authors, for example, Franklin,[31] Niederreiter,[32] Grothe,[33] and L'Ecuyer.[34] The characteristic polynomial of an MCG is defined as

$$f_B(x) = \det(x\mathbf{I} - \mathbf{B}) \bmod p. \tag{16}$$

An integer matrix $\mathbf{B} \in \mathbb{Z}_p^{k\times k}$ has order n if

$$n = \min_{j>0}\{j : \mathbf{B}^j \bmod p = I\}.$$

As a vector sequence $(\mathbf{X}_i)_{i\ge 0}$, an MCG has the maximum period of p^k-1 *if and only if* the matrix $\mathbf{B}$ has the order of p^k-1. It is well known that $\mathbf{B}$ has the order of p^k-1 *if and only if* its corresponding characteristic polynomial $f_{\mathbf{B}}(x)$ defined in Eq. (16) is a primitive polynomial. In particular, for every generator in MRG(k, p) that has a primitive characteristic polynomial, say, $f(x)$, the corresponding companion matrix $\mathbf{M}_f$ as in Eq. (13) has the order of p^k-1.

Theoretical Properties of MCGs

Consider the sequence of k-dimensional vectors generated by an MCG: $\mathbf{X}_1, \mathbf{X}_2, \cdots$. For a maximum period MCG, this vector sequence will repeat after the period of p^k-1 is reached. Therefore, any nonzero k-dimensional vector in $\mathbb{Z}_p^k$ will appear only once and this uniformity property over the k-dimensional cube of $\mathbb{Z}_p^k$ is a requirement for $\mathbf{X}_i/p$ to resemble a k-dim random vector uniformly distributed over $[0, 1]^k$. Additionally, any d-dim subvector $(d < k)$ from $\mathbf{X}_i$, $i = 1,2, \cdots$ will have the equi-distributional property over d-dimensional space which is similar to that of a maximum period MRG. That is, all d-dim nonzero subvectors will occur $p^{(k-d)}$ times and all d-dim zero subvectors will occur $p^{(k-d)}-1$ times. This 'column-wise' output method is particularly suitable for some vector processors to produce a sequence of successive k-dim random vectors over a high-dimensional space.

Using the Cayley-Hamilton Theorem, Grothe[33] first observed that, when taken as a k-dimensional vector sequence $(\mathbf{X}_i)_{i\ge 0}$, the MCG satisfies the same recursion as the generator in MRG(k, p) whose companion matrix $\mathbf{M}_f$ was used to define the matrix multiplier $\mathbf{B}$, that is, the vector sequence $(\mathbf{X}_i)_{i\ge 0}$ satisfies the following recursion:

$$\mathbf{X}_i = \alpha_1\mathbf{X}_{i-1} + \alpha_2\mathbf{X}_{i-2} + \cdots + \alpha_k\mathbf{X}_{i-k} \bmod p, \quad i \ge k. \tag{17}$$

Therefore, the k sequences taken from each of the k rows in Eq. (17) can be viewed as k copies of the same MRG with different starting seeds. In other words, as a vector sequence, a maximum-period MCG can be viewed as running k copies of the same generator in MRG(k, p) with different starting seeds. These can be viewed as k output streams corresponding to the k-rows in the generating equation in Eq. (17). As mentioned previously, each stream is produced by the same MRG sequence using the same generating equation with different starting shifts. Since the period length (p^k-1) is huge, random starting seeds can produce a reasonably large shift among k different streams. This 'row-wise' output method is particularly suitable for parallel simulation by assigning a different stream for each processor.

Therefore, it is important to consider the problem of choosing the matrix multiplier $\mathbf{B}$ with the characteristic polynomial $f(x)$ so that the corresponding MCG is efficient to implement in either 'column-wise' or 'row-wise' mode. Deng and Lin[35] proposed a general class of efficient MCGs (FMCGs) which can be as efficient as LCGs and Deng[21] provided some statistical justifications for MCGs.

Combination Generators for Small Order MRGs and LFSRs

Using a similar idea of combining LCGs, L'Ecuyer[36] suggested combining the following two 31-bit MRGs, each of order 3,

$$X_i = 63308X_{i-2} - 183326X_{i-3} \mod(2^{31}-1),$$

$$Y_i = 86098Y_{i-1} - 539608Y_{i-3} \mod(2^{31}-169),$$

to form a combined generator as

$$Z_i = (X_i - Y_i) \bmod (2^{31}-1), \quad U_i = Z_i/(2^{31}-1).$$

The U_i produced from the above combined generator is a very close approximation to

$$W_i = X_i/(2^{31}-1) + Y_i/(2^{31}-169) \mod 1,$$

which is equivalent to an MRG of order 3 with a large modulus and multipliers with large coefficients.[36] The period of this combined generator is about 7.78×10^{53}.

Another popular generator, called *MRG32k3a* was proposed by L'Ecuyer[37] as a combination of the two 32-bit MRGs:

$$X_i = 1403580X_{i-2} - 810728X_{i-3} \mod(2^{32}-209),$$

$$Y_i = 527612Y_{i-1} - 1370589Y_{i-3} \mod(2^{32}-22853).$$

Letting

$$Z_i = (X_i - Y_i) \mod(2^{32}-209)$$

the corresponding uniform (0,1) generator is

$$U_i = Z_i^*/(m_1+1), \quad m_1 = 2^{32}-209, \tag{18}$$

where $Z_i^* = Z_i$, if $Z_i > 0$ and $Z_i^* = m_1$, if $Z_i = 0$. The period length of *MRG32k3a* is about 3×10^{57}.

Clearly, the idea of combination generators is not limited to combining PRNGs of the same type. Marsaglia's Super-Duper generator was one of the earliest generators (see Learmonth and Lewis[38]) to combine two different types of PRNGs. The Super-Duper generator combined, by exclusive OR, an LCG and an LFSR to obtain a generator with period length around 2^{62} that performed better than either of its component generators alone. Super-Duper was used early on in several software packages, including S-PLUS and it is still an option in a random number generator function in R.

While the procedure of combining generators can generally improve the empirical performances of the PRNGs and increase the period length, it will decrease greatly the generating efficiency and it will not yield a PRNG with an exact high-dimensional equi-distributional property.

Issues in the Search for Maximal Period MRGs of Large Orders

Large order, maximum period MRGs have become very popular for their ability to rapidly generate high quality random variates due to their huge period lengths, equi-distribution for high dimensions, and excellent empirical performances. It is well known that the maximum period of an MRG of order k is p^k-1, which can be achieved if its characteristic polynomial

$$f(x) = x^k - \alpha_1 x^{k-1} - \cdots - \alpha_k \tag{19}$$

is a primitive polynomial. It is thus desirable to find the coefficients for MRGs of large orders in order to ensure the maximum period is attained. A set of necessary and sufficient conditions for $f(x)$ to be a primitive polynomial has been given in Knuth[5] (p. 30).

One of the major bottlenecks that slows down their algorithm in finding a large-order MRG involves factoring a huge number like p^k-1. L'Ecuyer et al.[39] proposed a method for bypassing this difficulty for $k \le 7$ and later L'Ecuyer[37] extended the method for $k \le 13$ that focused on finding p such that

$$R(k,p) = \left(p^k - 1\right)/(p-1) \tag{20}$$

is also a prime number. Such $R(k, p)$ is called a *generalized Mersenne prime (GMP)* in Deng.[40] A Mersenne prime is a prime of the form 2^k-1 (the popular MT19937 is based on a particular Mersenne prime with $k = 19937$). It remains an open problem whether there are *infinitely many* k's for which 2^k-1 is a prime. Currently, there are only 49 known Mersenne primes and only 15 Mersenne primes have been found since 1996. See, the Great Internet Mersenne Prime Search (GIMPS) at https://www.mersenne.org/. While the GMP is an extension of the Mersenne prime, the goal of GMPs is different. For a known prime k, we find p for which $R(k, p) = (p^k-1)/(p-1)$ is a prime (GMP). For a 32-bit RNG: for a prime k, we find c such that $p = 2^{31}-c$, and $R(k, p)$ are primes. Sometimes, we impose an additional condition on p that $Q = (p-1)/2$ is also a prime.

While it is hard to find Mersenne primes, it is relatively easy to find a GMP for any prime k.

Deng[41] found GMPs for k up to 10007 and Deng et al.[42] found GMPs for even larger values of k up to 25013. Deng et al.[43] found GMPs for some large k's with p of size 64-bit and 128-bit.

For 32-bit computers, it is common to choose prime modulus $p = 2^{31}-1$ because it is the largest (signed) integer that can be stored in a 32-bit computer word and its (expensive) modulus operation can be implemented with efficient logical operations as explained in Deng.[1] In the same paper, he studied an interesting question: What if we fix modulus $p = 2^{31}-1$ and search for prime order k such that $(p^k-1)/(p-1)$ is a prime? According to Deng,[1] there is *no* $k \le 60000$ for which $R(k, p)$ is a prime. However, he did find several k ($k = 47$, $k = 643$, $k = 1597$) such that $R(k, p) = q \times H$, where q is a product of several 'small' primes (say, $< 10^9$) and H is a (huge) prime. Therefore, such p^k-1 can be completely factorized. To characterize the possible factor of $R(k, p)$, Deng et al.[44] applied Theorem 3 below which is a special case of Legendre's Theorem for the prime factors of the form $a^k \pm b^k$:

Theorem 3. *Let k be a prime. For any prime factor q of $(p^k-1)/(p-1)$, $q = 1 \bmod 2k$.*

Therefore, any prime factor, say q, of $(p^k-1)/(p-1)$ is of the form $q = 2ck + 1$, for some integer c. When k is large, this result is useful to greatly speed up the search of possible factors by skipping all prime numbers $q \neq 1 \bmod 2k$ as possible factors. With the help of this result and an extensive computer search, Deng et al.[44] found the complete factorization of p^k-1 for $k = 7499$ and $k = 20897$.

With the complete factorization found, several efficient MRGs with modulus ($p = 2^{31}-1$) are found for orders 102 and 120 in Deng and Xu.[45] Several efficient and portable generators of order $k = 1597$ with period length of approximately $10^{14903.1}$ were constructed in Deng.[1] More recently, Deng et al.[44] further extended this work by finding DX and several other classes of generators of order $k = 7499$ and $k = 20897$. It remains an open question whether, with $p = 2^{31}-1$, there is *any* k such that $(p^k-1)/(p-1)$ is a GMP.

Deng[40] proposed an efficient search algorithm for finding maximal period MRGs of large order, called algorithm GMP, that provides an early exit strategy to overcome a second major bottleneck in the algorithm given by Knuth.[5]

Algorithm GMP

Let p be a prime, $f(x)$ be as in Eq. (19) and $R = (p^k-1)/(p-1)$.

1. $(-1)^{k-1}\alpha_k$ must be a primitive root mod p.
2. Initially, let $g(x) = x$. For $i = 1, 2, \ldots, \lfloor k/2 \rfloor$, where $\lfloor x \rfloor$ is the largest integer $\le x$, do
   (a) $g(x) = g(x)^p \bmod f(x)$
   (b) $d(x) = \gcd(f(x), g(x)-x)$
   (c) If $d(x) \neq 1$, then $f(x)$ cannot be a primitive polynomial. Exit.
3. For each prime factor q of R, the degree of $x^{R/q} \bmod (f(x), p)$ is positive.

If all three steps are completed with no early exit, then $f(x)$ is a primitive polynomial. Details of this algorithm along with the mathematical foundation are given in Deng[1] and Deng and Shiau.[46]

Algorithm GMP provides an efficient method for finding primitive polynomials due to an early exit strategy since a failed search is likely to terminate very early. To demonstrate this, we show a typical search result for an efficient MRG to be discussed later called a DX-k with $k = 3803$ using algorithm GMP. In total, the number of iterations needed for the successful search is 2858. For the 2857 failed searches, we tabulate the frequency of early exit for searching DX-3803 at the i-th iteration ($i = 1,2, \cdots, 1902$) in Table 1.

From Table 1, one can see that more than 90% of the failed searches will exit at $i \le 6$ iterations with the average number of iterations equal to 4.997. Because of the early exit, lots of computing time can be saved. Of course, the number of iterations needed until a successful search is 'random' but the example given is typical.

Compared with algorithm AK of Knuth,[5] algorithm GMP has been shown empirically to shorten search time as much as 1000-fold when the order of the MRG is large ($k > 5000$). Using algorithm GMP, many primitive polynomials of degree up to 25,013 corresponding to MRGs of periods up to $\approx 10^{233361}$ have been found; see, Deng et al.[42] The efficient algorithm GMP can also be applied to find MRGs for 64-bit and 128-bit computers as well, see Deng et al.[43] In recent years, great progress has been made in finding MRGs of large order.[1,40–42,44] With increases in computing power and drastic reductions in memory costs, the search for ever larger order MRGs with maximal periods and equi-distribution over higher dimensions is even more practical.

In summary, algorithm GMP has the following advantages:

1. It avoids the factorization of p^k-1 by searching for p such that $(p^k-1)/(p-1)$ is a prime number.

TABLE 1 | Frequency (Count) and Cumulative Percentage (Perc) of Early Exit for Searching DX-3803 at *i*-th Iteration

i	Count	Perc	*i*	Count	Perc	*i*	Count	Perc	*i*	Count	Perc
1	1787	.6253	20	4	.9724	39	2	.9867	111	2	.9941
2	399	.7649	21	2	.9731	40	1	.9871	112	1	.9944
3	192	.8320	22	3	.9741	41	1	.9874	124	1	.9948
4	97	.8660	23	4	.9755	44	1	.9878	128	2	.9955
5	77	.8929	24	3	.9766	45	1	.9881	134	1	.9958
6	46	.9090	25	2	.9773	48	1	.9885	145	1	.9962
7	32	.9202	26	5	.9790	50	1	.9888	163	1	.9965
8	20	.9272	27	1	.9794	54	1	.9892	170	1	.9968
9	20	.9342	28	2	.9801	55	1	.9895	173	1	.9972
10	18	.9405	29	3	.9811	56	1	.9899	176	2	.9979
11	20	.9475	30	1	.9815	58	1	.9902	217	1	.9982
12	11	.9514	31	3	.9825	59	2	.9909	255	1	.9986
13	10	.9549	32	1	.9829	62	1	.9913	296	1	.9990
14	10	.9584	33	2	.9836	63	1	.9916	563	1	.9993
15	9	.9615	34	1	.9839	68	1	.9920	587	1	.9996
16	12	.9657	35	1	.9843	74	1	.9923	1902	1	1.0000
17	5	.9675	36	1	.9846	90	1	.9927			
18	6	.9696	37	1	.9850	106	1	.9930			
19	4	.9710	38	3	.9860	108	1	.9934			

It is well known that the problem of primality testing is much easier than factorization. See, for example, Agrawal et al.[47]

2. It provides an early exit strategy for a failed search, which enables a quick exit in most cases except where a search is successful.

As CPU architectures are moving from 32 to 64 bits (or beyond), there is a need for new PRNGs that can increase the resolution of the simulated sequence. Greater precision may be of value in some of the more challenging research problems that arise. To meet this need, the search for large-order MRGs with prime modulus p of magnitude around the largest integer for a 64-bit or 128-bit word must be continued and expanded. The search algorithms for efficient multipliers in maximal order MRGs may need to be improved. Additional issues for the computer search for very-large-order MRGs are discussed in Deng.[41]

Portable and Efficient MRGs

The maximum period of an MRG of order k is p^k-1 which is much longer than the maximum period of an LCG of $p-1$. A general MRG may be computationally less efficient because several multiplications are required to compute the next number in the sequence whereas the LCG needs only one multiplication to produce the next result. To improve the efficiency of MRGs, Grube,[48] L'Ecuyer and Blouin,[49] and L'Ecuyer et al.[39] considered and provided portable implementations of MRGs in Eq. (11) with only two nonzero coefficients α_j and α_k $(1 \le j < k)$. As an alternative, Deng and Lin[35] proposed to set as many coefficients α_i in an MRG to 0 and/or ± 1 as possible. The resulting Fast MRG (FMRG) can be shown to be almost as efficient as a classical LCG. As another class of efficient generators, Deng and Xu[45] proposed the DX generator as a system of portable, efficient, and maximum-period MRGs, in which all the nonzero multipliers are the same. Deng[1] modified and extended the DX generators as follows:

1. DX-k-1-t ($\alpha_t = 1$, $\alpha_k = B$), $1 \le t < k$,

$$X_i = X_{i-t} + BX_{i-k} \bmod p, \quad i \ge k. \tag{21}$$

2. DX-k-2-t ($\alpha_t = \alpha_k = B$), $1 \le t < k$,

$$X_i = B(X_{i-t} + X_{i-k}) \bmod p, \quad i \ge k. \tag{22}$$

3. DX-k-3-t ($\alpha_t = \alpha_{\lceil k/2 \rceil} = \alpha_k = B$), $1 \le t < \lceil k/2 \rceil$

$$X_i = B\left(X_{i-t} + X_{i-\lceil k/2 \rceil} + X_{i-k}\right) \bmod p, \quad i \ge k. \tag{23}$$

4. DX-k-4-t ($\alpha_t = \alpha_{\lceil k/3 \rceil} = \alpha_{\lceil 2k/3 \rceil} = \alpha_k = B$),

$1 \le t < \lceil k/3 \rceil$,

$$X_i = B\left(X_{i-t} + X_{i-\lceil k/3 \rceil} + X_{i-2\lceil k/3 \rceil} + X_{i-k}\right) \bmod p, \quad i \ge k. \tag{24}$$

Here $\lceil x \rceil$ is the ceiling function of x, which is the smallest integer $\ge x$. We will call the generators thus constructed DX generators. To increase the generating efficiency, it is common to consider DX generators with the modulus $p = 2^{31}-1$ and some special multiplier, B, of the form $B = 2^r \pm 2^w$ for the DX generators. Such DX class of generators are as efficient as MT19937, and they can have period lengths much larger than the period of the popular MT19937.

Design of Efficient Large Order MRGs

L'Ecuyer[50] showed that the set of all d ($> k$) tuples in the generated MRG sequence as given in Eq. (11) are distributed in equidistant parallel hyperplanes at distance

$$D > \left(1 + \sum_{i=1}^{k} \alpha_i^2\right)^{-1/2}. \tag{25}$$

Therefore, a large $\sum_{i=1}^{k} \alpha_i^2$ is a necessary (but not sufficient) condition for a good MRG. Consequently, it is better to have more nonzero terms with larger α_i^2 in general. While this may give a nice justification for considering MRGs with many nonzero terms, we need also to consider their generating efficiency.

To construct a general class of efficient generators, consider a special class of MRGs that have at most two different nonzero coefficients, say, A and B. Let S_A and S_B be two index sets defined as $S_A = \{j \mid \alpha_j = A\}$ and $S_B = \{j \mid \alpha_j = B\}$, such that $S_A \cap S_B = \emptyset$. Deng et al.[51] proposed a general class of generators of the following form:

$$X_i = A\sum_{j \in S_A} X_{i-j} + B\sum_{j \in S_B} X_{i-j} \bmod p. \tag{26}$$

A simple way to make this class of generators computationally efficient is to have only few elements in both S_A and S_B. The previously defined DX generators are a special case of Eq. (26). DX generators, like any PRNGs with very few nonzero coefficients, typically have the drawback of a 'bad initialization effect.' This effect is observed when the k-dimensional state vector is close to the zero vector and the subsequent numbers generated tend to stay within a neighborhood of zero for many generations before they can break away from this near-zero state, a property not desirable in the sense of randomness. As a result of bad initialization, two generated sequences using the same DX generator with nearly identical state vectors may not depart from each other quickly enough. This 'bad initialization effect' was first observed by Panneton et al.[52] for MT19937.

Classes of MRGs can be designed to overcome this 'bad initialization effect' by including many nonzero terms, however, these MRGs tend to be inefficient. It is possible to find efficient implementations for certain generators by rewriting Eq. (26) as a simple higher-order recurrence equation, see Deng and Shiau[46] for details.

DL Generators

Examining the structure of generators of the form Eq. (26) with restrictions on the forms of S_A and S_B Deng et al.[51] propose a class of DL generators with $S_A = \{1,2, \ldots, t-1\}$ and $S_B = \{t, t+1, \ldots, k\}$ for $1 \le t < k$, and $g = 1$. Then

$$X_i = A(X_{i-1} + X_{i-2} + \cdots + X_{i-t+1}) + B(X_{i-t} + X_{i-t-1} + \cdots + X_{i-k}) \bmod p, \tag{27}$$

where t can be useful to expand the search space for maximal period DL generators.

An efficient implementation of DL generators is discussed in Deng and Shiau.[46]

DS Generators

Another set of generators that can be efficiently implemented were proposed by Deng et al.[51] Called DS generators, they also have many nonzero terms and are defined as:

$$X_i = B\sum_{j=1}^{k} X_{i-j} - CX_{i-d} \bmod p, i \ge k. \tag{28}$$

By introducing parameters B and C for the multipliers, and index d, the search parameter space is expanded. Like DL generators, DS generators can be efficiently implemented using a $(k+1)$st order recurrence equation, see Deng and Shiau.[46]

DT Generators

Another class of MRGs, called DT generators, with many nonzero terms and unequal weights on each term as discussed in Deng et al.[53] is given by

$$X_i = B^k X_{i-1} + B^{k-1} X_{i-2} + \cdots + B X_{i-k} \bmod p,\ i \geq k. \tag{29}$$

DT generators can also be efficiently implemented with a $(k + 1)$-order recurrence equation, see Deng and Shiau.[46]

Practical Implication of Bad Initialization Effect

All of the DX, DL, DS, and DT generators can have nice theoretical properties and are able to pass stringent empirical tests such as the battery of tests in TestU01, see Deng[1] for many choices of multipliers. It has been shown, however, that these generators will also have a 'bad initialization effect' as previously noted. It remains an open question on whether the 'bad initialization effect' will have a significant impact on the practical applications. Unless these seeds were deliberately chosen, it is extremely unlikely that two streams of random sequences would have nearly identical k-dimensional state vectors at any stage. The extremely rare occurrence of this event in practice makes the practical significance of this bad initialization effect unclear. Furthermore, although the DX generators are known to have a 'bad initialization effect' they still passed the stringent empirical tests, see, L'Ecuyer and Simard.[9] Still, a class of efficient generators that possess these excellent empirical properties that can move away from near-zero states quickly is desirable.

All of the DX, DL, DS, and DT generators enjoyed some statistical justifications as in Deng and George[19] and Deng et al.[20] Specifically, Deng and George[19] studied the statistical properties of generators under the assumption that the output sequences of component generators are finite-bit realizations of some independent random variables. In addition, Deng et al.[20] argued that an MRG is a certain form of a combined generator and provided some statistical properties and justifications of a general combined generator. Using a similar idea, Deng et al.[23] proposed a general method of TAC to improve the performance of some classical generators as an extension of the Wichmann and Hill generator.

Empirical Testing/Evaluation of PRNGs

Several empirical packages for testing a random number generator have been proposed including: (1) DIEHARD proposed in Marsaglia,[54] (2) NIST package and (3) TestU01 test suite which was developed by L'Ecuyer and Simard[9] with source code provided.

TestU01 is by far the most comprehensive test suite, which is composed of three predefined test modules: (1) *Small crush*: with 15 tests (2) *Crush*: with 144 tests (3) *Big crush*: the most comprehensive with 160 tests. Each of the tests will produce a p-value representing the probability of observing a test statistic that is more extreme than the one observed when the null hypothesis is true. At the end of each test, those p-values outside the range of $[10^{-3}, 1 - 10^{-3}]$ are reported by TestU01. In theory, the p-value for a test statistic is uniformly distributed over (0, 1) when the null hypothesis is true; see, for example, Murdoch et al.[55] Therefore, the p-values produced by a test for a 'perfect' uniform random number generator should follow the $U(0, 1)$ distribution. To judge the 'goodness' of the generators, we can evaluate the 'closeness' between the actual percentage of p-values produced and the expected values over various specified ranges.

Several popular PRNGs, such as MT19937, have been shown to consistently fail certain tests in TestU01 (L'Ecuyer and Simard[9]). In contrast DX and other related generators have consistently passed the same stringent TestU01; see L'Ecuyer and Simard[9] and Deng et al.[42] In addition L'Ecuyer and Simard[9] showed that all popular LFSRs and GFSRs failed the linear complexity test within their test library *TestU01*.

Pseudo Random Number Generators in the R Software System

The R software system has become increasingly popular and as such it is interesting to examine the PRNGs available for use in the base package of R. In the base package the random number generating function *RNGkind* has options for specifying a particular PRNG that include

1. 'Mersenne-Twister,' the MT19937 generator previously discussed
2. 'Wichmann-Hill' combination generator given in Eq. (5)
3. 'Marsaglia-Multicarry,' which is a multiply-with-carry generator as in Eq. (4)
4. 'Super-Duper' a previously mentioned combination generator combining LCG and LFSR as proposed by Marsaglia
5. 'Knuth-TAOCP-2002' which is a lagged Fibonacci generator as in Eq. (10)
6. 'Knuth-TAOCP' an older version of 'Knuth-TAOCP-2002' with a different initialization

7. 'L'Ecuyer-CMRG' (MRG32k3a) a combination generator combining two MRGs of order 3 as given in Eq. (18)

As a comparison among the above PRNGs used in R, we note that

1. MT19937 has the longest period length of $2^{19937}-1 \approx 10^{6001}$ and it is the only PRNG that can claim an exact equi-distribution up to 623 dimensions. MT19937 is very efficient although it is not the fastest among the group. See, L'Ecuyer and Simard.[9]
2. Knuth-TAOCP can also be considered as a special MRG of order 100 with a nonprime modulus 2^{30} and small multipliers (± 1) which has a much shorter period length when compared with maximal-period MRGs. No equi-distribution can be claimed for Knuth-TAOCP. It is a fast generator since no multiplication operation is required; however, it requires a long and careful initialization. For example, we need to make sure not all (or most) of the 100 internal states of X_i in Eq. (10) are even numbers.
3. MRG32k3a is the only one that can pass stringent batteries of tests in TestU01 as shown in L'Ecuyer and Simard.[9] It also has an efficient implementation for parallel simulation due to the use of small order MRGs. However, MRG32k3a and the Wichmann-Hill generator are also among the slowest PRNGs.

The default generator, if none is specified, is MT19937. To implement other PRNGs, the *RNGkind* function also has a 'user-specified' option.

Comparisons of PRNGs in R with DX Generators

DX and other classes (DL, DS, and DT) of generators are special classes of MRGs which are as efficient as LCGs with much longer period length and nice equi-distributional property over high-dimensional space. Many DX and other classes of generators have been found with a wide range of choices of order k (up to 25013 with period length of 10^{233361}) and with various sizes/types of the modulus p (32-bit, 64-bit, 128-bit, or beyond). Selecting parameter choices for efficiency considerations, i.e., choosing the prime modulus of $p = 2^{31}-1$ and specialized multiplier $B = 2^r \pm 2^w$, the largest order of the DX-k generators found is $k = 20897$ with period length of $10^{195009.3}$. Some of the smaller order DX-k generators, DX-47 and DX-1597 (with period length $10^{14903.1}$) have been shown to pass the stringent Big Crush battery of tests in TestU01, see L'Ecuyer and Simard.[9] In contrast, except for L'Ecuyer-CMRG (MRG32k3a), no other PRNGs (including MT19937) listed in the R generator options can pass the same empirical tests.

In terms of generating efficiency, DX generators are quite efficient. For a general multiplier, L'Ecuyer and Simard[9] show that a DX generator is about twice as fast as the MRG32k3a ('L'Ecuyer-CMRG') and, with efficient implementation, the DX generator can be up to five times as efficient as MRG32k3a.[44] These specialized DX generators are also faster than MT19937. In addition, DX-k generators have by far the longest period length and the property of equi-distribution over high dimensional space. Except for MT19937 and DX generators, none of the PRNG options in *RNGkind* can claim an exact equi-distribution property over any high dimensional space.

One nice feature for L'Ecuyer-CMRG (MRG32k3a) is that it can be used in the package *parallel* for parallel random number generation. We show next there are new and efficient parallel simulation procedures using DX generators as baseline generators.

Parallel Random Number Generators

As mentioned in the introduction, one of the issues that may be considered when selecting or comparing PRNGs is the ability to parallelize the generator and to quickly create substreams that are (practically) independent. To build a general-purpose discrete-event simulator that can be run on parallel computers, a good parallel random number generator is required. An essential component of a good parallel PRNG is a method of generating 'independent streams' of random numbers. It is possible to achieve this by either choosing PRNGs with different (a) moduli, (b) additive constants, (c) starting seeds, or (d) multipliers for different processors. For more details on various methods proposed, see Deng et al.,[4] Mascagni,[56] Mascagni and Srinivasan,[57] Srinivasan et al.,[58] and L'Ecuyer et al.[59] For many of the parallel PRNGs a fixed block method is used to assign substreams to separate processors. This requires a (efficient) means of skipping ahead in the generator sequence to provide adjacent segments that do not overlap.

LCGs and MRGs have served as baseline generators for some parallel random number generators. For LCGs, Deng et al.[4] proposed a systematic leap-frog method to automatically choose different multipliers for the corresponding LCGs so that they are of

the maximum period. For MRGs, Deng[40] proposed an automatic generation method to construct many maximum-period MRGs from the baseline MRG for processors in the parallel simulation.

Parallel Random Number Generators Using LCGs

Deng et al.[4] proposed to generate a sequence of random numbers using an LCG such that every generated number r is relatively prime to $p-1$ for p chosen to maximize the period of the LCG. The method is described as: first choose a multiplier, R, that is a primitive element modulo $p-1$ and is relatively prime to $p-1$; then, generate a sequence of such r's using recursive equation:

$$r_n = R r_{n-1} \bmod (p-1), \; n \geq 1, \tag{30}$$

where r_0 is any nonzero initial seed that is relatively prime to $p-1$; $r_0 = 1$ for simplicity. Using Eq. (30), distinct r_n's (within the period of the LCG in Eq. (30)) can be produced that are relatively prime to $p-1$. These values can be used to assign each processor in a parallel environment its own LCG using the systematic leapfrog method proposed by Deng et al.[4] The algorithm for the systematic leapfrog is summarized as follows:

Algorithm for LCG Systematic Leapfrog

Suppose B and R are the multipliers of the LCG for the first processor and the baseline LCG in Eq. (30), respectively,

1. When initiating a new processor, generate a new r using the following equation:

$$r_{\text{new}} = R r_{\text{old}} \bmod (p-1).$$

2. Calculate a multiplier B_{new} for the new processor as

$$B_{\text{new}} = B^{r_{\text{new}}} \bmod p.$$

3. The new processor is assigned LCG with multiplier B_{new} as

$$X_i = B_{\text{new}} X_{i-1} \bmod p, \; i \geq 0.$$

Examples of the LCG systematic leapfrog are given in Deng and Shiau.[46] As another larger example to show the effect of correlation among various processors, we choose LCG(B, p) with $B = 5$ and $p = 107$. For the fixed block method, we choose five initial seeds X_0 so that they are roughly far apart from each other. For the systematic leapfrog method, we choose five different multipliers B each resulting in LCGs with the maximum period. The pairwise scatter plots for the two methods are shown in Figure 4.

From Figure 4(a), one can see that there are clear patterns of linear relations between the variates generated from different streams for the fixed block method. This problem of a high correlation among different streams was first reported in Anderson and Titterington.[60] On the other hand, no patterns can be observed among different streams for the systematic leap frog method as shown in Figure 4(b).

As an example, let the prime modulus $p = 2^{31}-69 = 2147483579$ and specify the two multipliers as $B = 1747834819$ (a primitive element modulo p) and $R = 693352593$ (a primitive element modulo $p-1$). Here the modulus p is chosen so that $p-1 = 2Q$ for $Q = 1073741789$ also a prime number. For initial value $r_{\text{old}} = 1$, a new r is generated for a new processor by

$$r_{\text{new}} = 693352593 r_{\text{old}} \bmod 2147483578.$$

When $r_{\text{old}} = 1$, $r_{\text{new}} = 693352593$. Next, calculate a new B as

$$B_{new} = 1747834819^{r_{new}} \bmod 2147483579 = 315852573.$$

The new LCG generator for the new processor is given by:

$$X_i = 315852573 X_{i-1} \bmod 2147483579, \; i \geq 1.$$

Repeating this procedure, $Q-1 = 1073741788$ distinct LCGs can be quickly and randomly constructed, each with the maximum period of 2147483578.

The 'systematic leap frog method' is superior to the 'fixed block method' for specifying PRNGs for different processors because under the leapfrog method each processor has its own LCG and so the potential overlapping problem of the generated LCG sequences does not exist. Furthermore, the correlation and linear structure problems are greatly minimized as shown in Figure 4. While the leapfrog method minimizes the problem of 'within processor correlation' with different LCGs for different processors, the problem of 'within processor correlation' (with the same LCG) may still exist however. Next, we discuss a natural way to minimize the within processor correlation by replacing the LCG with a large-order MRG as the baseline generator.

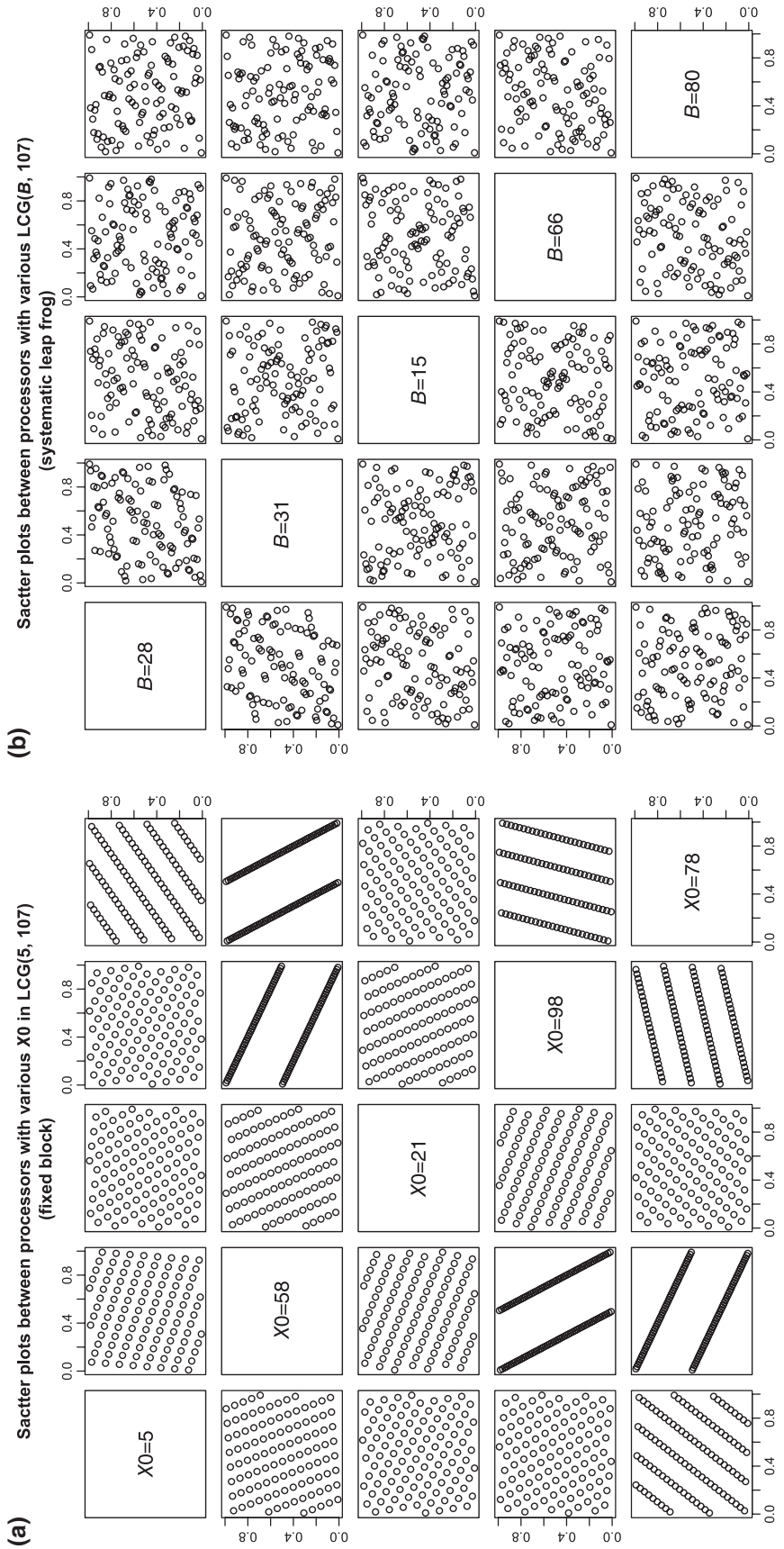


FIGURE 4 | Scatter plots of correlation between pairs of five processors. Panel (a) is for the fixed block method and panel (b) is for the systematic leap frog method.

Parallel Random Number Generators Using MRGs

Deng[40] proposed an efficient method for finding large-order, maximum period MRGs. He also described a method for constructing maximum-period MRGs from a single MRG for DX generators in particular. Denote by MRG-*k*-*s* the class of maximum-period MRGs of order *k* with *s* nonzero coefficients in their corresponding characteristic polynomials. When $R(k, p) = (p^k-1)/(p-1)$ is a prime, Deng[40] and Deng et al.[61] proposed a simple method to construct MRG-*k*-*s* generators from the characteristic polynomial of a DX-*k*-*s* generator. Using an idea similar to the systematic leap frog method, Deng[40] proposed an automatic generating method (AGM) for finding values of *c* such that

$$G(x) = c^{-k} f(cx) \bmod p \tag{31}$$

is a primitive polynomial, where $f(x)$ is the characteristic polynomial of the MRG-*k*-*s* as in Eq. (19). The new characteristic equation in Eq. (31) can in turn define an MRG with a different structure from the MRG defined by $f(x)$; see Deng[40] and Deng et al.[61] for a detailed discussion. Algorithm AGM, discussed below, was proposed in Deng and Shiau.[46]

Algorithm AGM

Let R be a primitive element modulo $p-1$ such that $\gcd(R, p-1) = 1$. The following procedure will randomly generate a sequence of maximum-period MRGs:

1. When a new processor is initiated, generate r_n by the following equation:

$$r_n = R r_{n-1} \bmod (p-1), n \geq 1 \text{ with } r_0 = 1.$$

2. Calculate d_n and c_n for the new processor by

$$d_n = k^{-1}(r_n + 1) \bmod (p-1) \text{ and } c_n = B^{d_n} \bmod p.$$

3. With the given c_n, compute the primitive polynomial $G(x)$ by

$$G(x) = c_n^{-k} f(c_n x) \bmod p.$$

4. The new processor is assigned the newly constructed maximum-period MRG corresponding to the characteristic polynomial $G(x)$ as follows:

$$X_i = G_1 X_{i-1} + \cdots + G_k X_{i-k} \bmod p.$$

Using algorithm AGM many MRGs (up to 100,000/second) with maximum period may be quickly found. Each of the MRGs found can be assigned to one of the subtasks or processors in a parallel simulation. Deng et al.[53] extended this approach to generate different MRGs 'randomly,' quickly, and automatically, while maintaining the maximum-period property.

Comparison of Parallel Random Number Generators

To perform computational tasks in parallel, each of a number of processors must be able to generate a unique stream of pseudo-random numbers that are independent of streams from other processors. Otherwise, even if each processor is given a unique starting seed, there would be a chance for streams generated by any two processors to overlap at some point. This is particularly true if the period length of the PRNG is small or the number of multiprocessors is large. Overlapping streams of random numbers would introduce the potential for long-range correlation between different processors, jeopardizing the validity of simulations or Monte Carlo results.

We believe that it is a good idea to use different MRGs (via AGM algorithm) for different processors so that we can minimize the possible (unknown) 'defects' on a fixed generator for certain applications. As an example, if we were to conduct a coin-tossing experiment to generate 1,000,000 bits. We can either select a (hopefully unbiased) coin and toss it 1,000,000 times or we can ask 1000 persons to select and toss 1000 different coins. Clearly, the first procedure is not recommended because it relies heavily on the assumption that a fair coin was selected and it will take a much longer time to complete the task.

One naive method of jumping ahead is simply to multiply the generator's state-space by a precomputed matrix of a very large order. However, this operation would require excessive memory space and can be very time-consuming to perform. Using MT19937, a number of methods for 'jumping ahead' in the generated sequence to obtain different starting seeds have been proposed. See, Haramoto et al.[62,63] The main advantage of the automatic generating methods described earlier is that each processor can be (quickly) assigned with a distinct maximum-period MRG. Therefore, the process does not suffer from the potential problem of long-range correlation between different processors.

CONCLUSIONS

There are several advantages for choosing large order MRGs as random number generators in the area of computer simulation: (1) many large order MRGs have been found with the help of the

efficient algorithm GMP; (2) they have the known theoretical properties of long period length (p^k–1) and high-dimensional equi-distribution; (3) there are several efficient classes of generators, like DX generators, which are as efficient as LCGs and the popular MT19937; (4) several efficient classes of MRGs have excellent performances, passing stringent empirical tests; and (5) there is an efficient implementation for the automatic generation of MRGs suitable for a large scale parallel simulation.

FURTHER READING

Brillhart J, Lehmer DH, Selfridge JL, Tuckerman B, Wagstaff SS. *Factorizations of* $b^n \pm 1$, b=*2,3,5,6,7,10,11,12 Up to High Powers*. 3rd ed. Providence, RI: American Mathematical Society; 2002.

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

Williams HC. *Edouard Lucas and Primality Testing*. New York, NY: John Wiley & Sons; 1998.

Williams HC, Seah E. (a^n–1)/(a–1). *Math Comput* 1979, 33:1337–1342.

REFERENCES

1. Deng L-Y. Efficient and portable multiple recursive generators of large order. *ACM Trans Model Comput Simul* 2005, 15:1–13.
2. Lehmer DH. Mathematical methods in large-scale computing units. In: *Proceedings of the 2nd Symposium on Large-Scale Digital Calculating Machinery*, 141–146, Harvard University Press, Cambridge, MA, 1951.
3. Marsaglia G. The structure of linear congruential sequences. In: Zaremba SK, ed. *Applications of Number Theory to Numerical Analysis*. New York, NY: Academic Press; 1972, 249–286.
4. Deng L-Y, Chan KH, Yuan Y. Random number generators for multiprocessor systems. *Int J Model Simul* 1994, 14:185–191.
5. Knuth DE. *The Art of Computer Programming, Vol 2: Seminumerical Algorithms*. 3rd ed. Boston, MA: Addison-Wesley; 1998.
6. Park SK, Miller KW. Random number generators: good ones are hard to find. *Commun ACM* 1988, 31:1192–1201.
7. Wu P-C. Multiplicative, congruential random-number generators with multiplier $\pm 2^{k_1} \pm 2^{k_2}$ and modulus 2^p–1. *ACM Trans Math Softw* 1997, 23:255–265.
8. Marsaglia G. Random numbers fall mainly in the planes. *Proc Natl Acad Sci U S A* 1968, 61:25.
9. L'Ecuyer P, Simard R. Testu01: a C library for empirical testing of random number generators. *ACM Trans Math Softw* 2007, 33:22.
10. Marsagila G. A random number generator for C, 1997. Available at: http://mathforum.org/kb/thread.jspa?forumID=231&threadID=496664&messageID=1518933#1518933. (Accessed July 14, 2017).
11. Marsaglia G, Zaman A. A new class of random number generators. *Ann Appl Prob* 1991, 1:462–480.
12. Couture R, L'Ecuyer P. Distribution properties of multiply-with-carry random number generators. *Math Comput* 1997, 66:591–607.
13. Goresky M, Klapper A. Efficient multiply-with-carry random number generators with maximal period. *ACM Trans Model Comput Simul* 2003, 13:1–12.
14. Wichmann B, Hill D. An efficient and portable pseudo-random number generator. *J R Stat Soc Ser C Appl Stat* 1982, 31:188–190.
15. Zeisel H. Remark AS R61: a remark on algorithm as 183. An efficient and portable pseudo-random number generator. *J R Stat Soc Ser C Appl Stat* 1986, 35:89.
16. Gentle JE. *Random Number Generations and Monte Carlo Methods*. 2nd ed. New York, NY: Springer-Verlag; 2003.
17. Brown M, Solomon H. On combining pseudorandom number generators. *Ann Stat* 1979, 7:691–695.
18. Marsaglia G. A current view of random number generators. In: *Computer Science and Statistics, Sixteenth Symposium on the Interface*, Elsevier Science Publishers, North-Holland, Amsterdam, 3–10, 1985.
19. Deng L-Y, Olusegun George E. Generation of uniform variates from several nearly uniformly distributed variables. *Commun Stat Simul Comput* 1990, 19:145–154.
20. Deng L-Y, Lin DKJ, Wang J, Yuan Y. Statistical justification of combination generators. *Stat Sinica* 1997, 7:993–1003.
21. Deng L-Y. Recent developments on pseudo-random number generators and their theoretical justifications. *J Chin Stat Assoc* 2016, 54:154–179.
22. Deng L-Y, Chu Y-C. Combining random number generators. In: *Proceedings of the 23rd Conference on Winter Simulation*, WSC '91, 1043–1046, Washington, DC, USA, 1991. IEEE Computer Society.

ISBN 0-7803-0181-1. Available at: http://dl.acm.org/citation.cfm?id=304238.304413. (Accessed July 14, 2017).

23. Deng L-Y, Guo R, Lin DKJ, Bai F. Improving random number generators in the Monte Carlo simulations via twisting and combining. *Comput Phys Commun* 2008a, 178:401–408.
24. Watson EJ. Primitive polynomials (mod 2). *Math Comp* 1962, 16:368–369.
25. Lewis TG, Payne WH. Generalized feedback shift register pseudorandom number algorithms. *J ACM* 1973, 20:456–468.
26. Coddington PD. Analysis of random number generators using Monte Carlo simulation. *Int J Mod Phys* 1994, C5:547.
27. Matsumoto M, Kurita Y. Twisted GFSR generators. *ACM Trans Model Comput Simul* 1992, 2:179–194.
28. Matsumoto M, Nishimura T. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans Model Comput Simul* 1998, 8:3–30.
29. Marsland EG. *Machine Learning*. Boca Raton, FL: CRC Press; 2011.
30. Lidl R, Niederreiter H. *Introduction to _nite _elds and their applications*. Cambridge, England: Cambridge University Press; 1994.
31. Franklin JN. Equidistribution of matrix-power residues modulo one. *Math Comput* 1964, 18:560–568.
32. Niederreiter H. A pseudorandom vector generator based on finite field arithmetic. *Math Japonica* 1986, 31:759–774.
33. Grothe H. Matrix generators for pseudo-random vector generation. *Statistische Hefte* 1987, 28:233–238.
34. L'Ecuyer P. Random numbers for simulation. *Commun ACM* 1990, 33:85–97.
35. Deng L-Y, Lin DKJ. Random number generation for the new century. *Am Stat* 2000, 54:145–150.
36. L'Ecuyer P. Combined multiple recursive random number generators. *Oper Res* 1996, 44:816–822.
37. L'Ecuyer P. Good parameters and implementations for combined multiple recursive random number generators. *Oper Res* 1999, 47:159–164.
38. Learmonth GP, Lewis PAW. Statistical tests of some widely used and recently proposed uniform random number generators. In: Kennedy WJ, ed. *Computer Science and Statistics: 7th Annual Symposium on the Interface*. Ames, IA: Statistical Laboratory, Iowa State University; 1973, 163–171.
39. L'Ecuyer P, Blouin F, Couture R. A search for good multiple recursive random number generators. *ACM Trans Model Comput Simul* 1993, 3:87–98.
40. Deng L-Y. Generalized Mersenne prime number and its application to random number generation. In: *Monte Carlo and Quasi-Monte Carlo Methods 2002*. New York, NY: Springer; 2004, 167–180.
41. Deng L-Y. Issues on computer search for large order multiple recursive generators. In: *Monte Carlo and Quasi-Monte Carlo Methods 2006*. New York, NY: Springer; 2008, 251–261.
42. Deng L-Y, Shiau J-JH, Lu HH-S. Efficient computer search of large-order multiple recursive pseudo-random number generators. *J Comput Appl Math* 2012a, 236:3228–3237.
43. Deng L-Y, Lu HH-S, Chen T-B. 64-bit and 128-bit dx random number generators. *Computing* 2010, 89:27–43.
44. Deng L-Y, Shiau J-JH, Lu HH-S. Large-order multiple recursive generators with modulus 2^{31}–1. *INFORMS J Comput* 2012b, 24:636–647.
45. Deng L-Y, Xu H. A system of high-dimensional, efficient, long-cycle and portable uniform random number generators. *ACM Trans Model Comput Simul* 2003, 13:299–309.
46. Deng L-Y, Shiau J-JH. Uniform random numbers. In: *Wiley StatsRef: Statistics Reference Online*. Hoboken, NJ: Wiley; 2015, 1–14.
47. Agrawal M, Kayal N, Saxena N. Primes is in p. *Ann Math* 2004, 160:781–793.
48. Grube A. Mehrfach rekursiv-erzeugte pseudo-zufallszahlen. *J Appl Math Mech* 1973, 53:T223–T225.
49. L'Ecuyer P, Blouin F. Linear congruential generators of order $k > 1$. In: *Winter Simulation Conference: Proceedings of the 20th Conference on Winter Simulation*, vol. 12, 432–439, 1988.
50. L'Ecuyer P. Bad lattice structures for vectors of non-successive values produced by some linear recurrences. *INFORMS J Comput* 1997, 9:57–60.
51. Deng L-Y, Li H, Shiau J-JH, Tsai G-H. Design and implementation of efficient and portable multiple recursive generators with few zero coefficients. In: *Monte Carlo and Quasi-Monte Carlo Methods 2006*. New York, NY: Springer; 2008b, 263–273.
52. Panneton F, L'ecuyer P, Matsumoto M. Improved long-period generators based on linear recurrences modulo 2. *ACM Trans Math Softw* 2006, 32:1–16.
53. Deng L-Y, Shiau J-JH, Tsai G-H. Parallel random number generators based on large order multiple recursive generators. In: *Monte Carlo and Quasi-Monte Carlo Methods 2008*. New York, NY: Springer; 2009a, 289–296.
54. Marsaglia G. The Marsaglia random number CDROM including the DIEHARD battery of tests of randomness. Available at: http://stat.fsu.edu/pub/diehard, 1996.
55. Murdoch D, Tsai Y-L, Adcock J. *P*-values are random variables. *Am Stat* 2008, 62:242–245.
56. Mascagni M. Parallel linear congruential generators with prime moduli. *Parallel Comput* 1998, 24:923–936.

57. Mascagni M, Srinivasan A. Algorithm 806: Sprng: a scalable library for pseudorandom number generation. *ACM Trans Math Softw* 2000, 26:436–461.

58. Srinivasan A, Mascagni M, Ceperley D. Testing parallel random number generators. *Parallel Comput* 2003, 29:69–94.

59. L'Ecuyer P, Simard R, Jack Chen E, David Kelton W. An object-oriented random-number package with many long streams and substreams. *Oper Res* 2002, 50:1073–1075.

60. Anderson NH, Titterington DM. Cross-correlation between simultaneously generated sequences of pseudo-random uniform deviates. *Statist Comput* 1993, 3:61–65.

61. Deng L-Y, Li H, Shiau J-JH. Scalable parallel multiple recursive generators of large order. *Parallel Comput* 2009b, 35:29–37.

62. Haramoto H, Matsumoto M, L'Ecuyer P. A fast jump ahead algorithm for linear recurrences in a polynomial space. In: *Proceedings of the 5th International Conference on Sequences and their Applications, SETA 08*, 290–298, 2008a.

63. Haramoto H, Matsumoto M, Panneton P, L'Ecuyer P. Efficient jump ahead for F_2-linear random number generators. *INFORMS J Comput* 2008b, 20:385–390.

