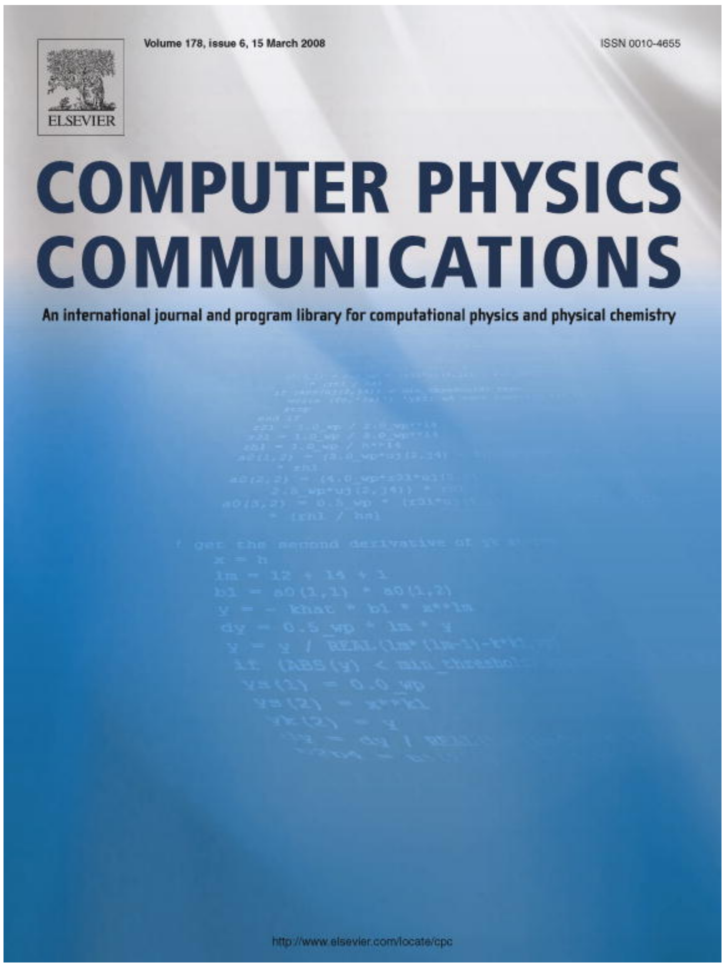
Volume 178, issue 6, 15 March 2008
ISSN 0010-4655
ELSEVIER
COMPUTER PHYSICS COMMUNICATIONS
An international journal and program library for computational physics and physical chemistry
http://www.elsevier.com/locate/cpc

Available online at www.sciencedirect.com

ScienceDirect

Computer Physics Communications 178 (2008) 401–408

Computer Physics Communications

www.elsevier.com/locate/cpc

Improving Random Number Generators in the Monte Carlo simulations via twisting and combining

Lih-Yuan Deng [a], Rui Guo [b], Dennis K.J. Lin [c], Fengshan Bai [b,*,1]

[a] *Department of Mathematical Sciences, University of Memphis, Memphis, TN 38152, USA*
[b] *Department of Mathematical Sciences, Tsinghua University, Beijing, 100084, PR China*
[c] *Department of Supply Chain and Information Systems, Pennsylvania State University, University Park, PA 16802, USA*

Received 8 March 2007; received in revised form 21 September 2007; accepted 19 October 2007

Available online 24 October 2007

Abstract

Problems for various random number generators accompanying the Wolff algorithm [U. Wolff, Phys. Rev. Lett. 62 (1989) 361; U. Wolff, Phys. Lett. B 228 (1989) 379] are discussed, including the hidden errors first reported in [A.M. Ferrenberg, D.P. Landau, Y.J. Wong, Phys. Rev. Lett. 69 (1992) 3382]. A general (though simple) method of twisting and combining for improving the performance of these generators is proposed. Some recent generators motivated by such a twisting and combining method with extremely long period are discussed. The proposed method provides a novel and simple way to improve RNGs in its performance.

PACS: 75.40.Mg; 05.70.Jk; 64.60.Fr

Keywords: DX generator; Hidden errors; LCG; LFG; MRG; RANLUX; SWC; Twisting and Combining (TAC); Wolff algorithm

1. Wolff algorithm and random number generators

1.1. Wolff algorithm

Monte Carlo simulations have become a standard practice in many scientific research including the field of computational physics. In particular, Wolff algorithm, proposed in [1,2], is an efficient cluster-flipping algorithm for the Ising model. It has been very popular in the area of statistical mechanics simulations. The Wolff algorithm employs a random number generator to decide whether or not to flip another spin. The probability of flipping another spin is a fixed probability which is called the transition probability. Clearly, the performance of the Wolff algorithm depends on that of the random number generator used. The existence of "hidden errors" is reported in [3]. This typically occurs when Wolff algorithm is companying several popular random number generators. Since then, there are many analytical and empirical study on the possible deficiency of these classical generators [4–17]. Most of them were concentrated on the mathematical analysis on the problem identifications of specific generators. Not much work has been done on the solution, however. That is, how to avoid the hidden problem as in the Wolff algorithm, or in computer simulation in general?

In what follows, we first give detailed description and then identify the common features and problems of various generators. In addition, we discuss their connections with a general class of generators, called Multiple Recursive Generators (MRGs). Section 2 introduces a novel and simple method to improve the performance of RNGs by "twisting and combining" (TAC) several generators. Next we discuss a special class of MRG, called DX generators, which are portable and efficient. Besides its high efficiency, DX generators also entertain the nice theoretical properties of MRGs. The main motivation behind DX generators is using the TAC method. Section 3 compares the general performances of various generators through simulations on the Wolff algorithm. In Section 4, these generators are evaluated through extensive statistical tests. It is shown that the proposed TAC is an instructive, simple and efficient method to improve the performance of existing RNGs. Section 5 gives a brief summary and conclusion.

[*] Corresponding author.
E-mail address: fbai@math.tsinghua.edu.cn (F. Bai).
[1] Supported partly by National Science Foundation of China 10501030.

1.2. *Linear Congruential Generator (LCG)*

The linear congruential generator, proposed in [18], has been the most commonly used pseudo-random number generator. A sequence of random numbers is obtained by setting

$$x_n = Bx_{n-1} \bmod m, \quad n \geqslant 1,$$

where x_n, B and m are non-negative integers and x_0 is a nonzero seed. The quality of the generator is determined by the choice of multiplier B, and modulus m; denoted by LCG($B; m$). It is common to choose the modulus m as a large prime number and the multiplier B to achieve the maximum period of $m - 1$. Such multiplier B is a primitive root over a finite field of m elements.

The most famous LCG is LCG(16807; $2^{31} - 1$), with $B = 16807$ and $m = 2^{31} - 1$. Noting that in [3], LCG(16807; $2^{31} - 1$) was called CONG. To improve the generating efficiency, a special form for the multiplier was proposed in [19], by taking

$$B = \pm 2^p \pm 2^q$$

with the modulus $m = 2^{31} - 1$.

LCG($B; m$) with $B = 2^{15} - 2^{10} = 31744$ was suggested in [19]. It results in a fast computation by using only shift and addition operations. In terms of computing time required, operations such as multiplication, division, or modulus are much more expensive than the operations of addition, subtraction or logical arithmetics. While this general class of generators is efficient, it may not produce sufficient mixing among bits of successive numbers generated. For example, it was pointed out in [20] that the successive values in the output of LCGs show strong dependence on corresponding Hamming weights.

1.3. *GFSR*

It was proposed in [21] that a more efficient method known as the generalized feedback shift-register (GFSR) in which numbers are formed by the same kth order recursive relation via exclusive-or ($\oplus$) operation. An efficient computer program for faster initialization and GFSR's was developed in [22]. Because of their generating efficiency, the following generators become very popular in the field of computational physics:

R250: $x_n = x_{n-250} \oplus x_{n-103}$; and

R1279: $y_n = y_{n-1279} \oplus y_{n-1063}$.

Here, x_n and y_n are 32-bits integers that are computed iteratively form previous numbers generated. The number of initial seeds required are 250 and 1279 for R250 and R1279, respectively. It is common to produce the required seeds by other random number generator such as LCG(16807; $2^{31} - 1$).

The period lengths for R250 and R1279 are $2^{250} - 1$ ($\approx 10^{75.3}$) and $2^{1279} - 1$ ($\approx 10^{385.1}$), respectively. While R250 and R1279 are efficient with a long period length, their empirical performances had some problems as reported by [3].

To consider a variation of GFSR, it is common to replace exclusive-or ($\oplus$) operation with simply a $+$ or $-$ operation and modulus m operation. Therefore, we have a Lagged Fibonacci Generator (LFG):

$$x_n = (x_{n-q} \pm x_{n-p}) \bmod m.$$

This generator is denoted as LFG($p, q; m$). LFG is fast because no multiplication operation is required. However, it has several drawbacks: (1) it has a bad lattice structure because of its small coefficients, (2) its empirical performance is generally poor, and (3) it is hard to find p, q and m to maximize the period length of the generator.

1.4. *Marsaglia and Zaman's generators*

Another type of generators studied in [3] is based on "subtract-with-borrow (carry)" (SWB/SWC) and "add-with-carry" (AWC) methods which are proposed by [23–26]. They are simple modifications of the Lagged Fibonacci Generator by including a carry from previous subtraction or addition operation:

$$x_n = x_{n-q} - x_{n-p} - c_n \bmod m,$$

where $c_1 = 0$, and $c_n = 0$, if $x_{n-q} - x_{n-p} - c_{n-1} < 0$; $c_n = 1$, otherwise. Such generators are referred as SWB generators or SWC generators. We will use the term SWC as in [3]. AWC generators can be similarly defined, though they will not be further discussed here.

The choice of p, q, m determines the period of the SWC generators. As shown in [26], the period of SWC($p, q; m$) generator is $\phi(m^p - m^q + 1)$ (see also [27], Theorem 8.2.6, p. 367), where $\phi(x)$ is the Euler "totient" function of x, which is the number of integers between 1 and x that are relatively prime to x. Hence, if $M = m^p - m^q + 1$ is a prime number, then the generator has a maximum period of $m^p - m^q$.

Generally speaking, SWC generators can be very efficient because no multiplication is required. In particular, the SWC(43, 22; $2^{32} - 5$) generator, is called SWC in [3], and its period is $10^{414.2}$. This kind of generator is slightly more complicated than the LFG and it is almost as efficient. However, similar problems of its empirical performance is also reported in [3].

1.5. *RANLUX (Luxury RNG)*

Another SWC generator frequently studied is SWC(24, 10; 2^{24}) which has a period length about 5.15×10^{171}. Analytical study on improving SWC(24, 10; 2^{24}) is given in [10] and a program provided by [11] to implement RANLUX generators by dropping certain numbers. Using LUX = 0, 1, 2, 3, 4 to control different level of discarding of SWC(24, 10; 2^{24}), one can choose 24 (out of p) random numbers then discarding p-24 random numbers where $p - 24 = 0$ (LUX = 0), 24 (LUX = 1), 73 (LUX = 2), 199 (LUX = 3) and 365 (LUX = 4). RANLUX has become very popular in the computational physics literature. The major advantage of RANLUX is that it has some theoretical support to improve the performance of the SWC generator. According to the empirical tests results reported in [12], RANLUX performed very well. The original version of RANLUX (with 24-bit precision) was not reported in [28] because some of the tests used only are applicable to generators with 32-bit (or higher) precision. Instead, [28] considered some decimation method on RANLUX with a large luxury level. Specifically, one can increase the bits of precision obtained from 24 to 48 by adding two successive numbers (from a sequence produced by RANLUX) modulo 1 as in

$$u_i = (x_{2i}/2^{24} + x_{2i+1}/2^{48}) \bmod 1, \quad i = 0, 1, 2, \ldots.$$

While the modified version of RANLUX has 48 bits of precision and very robust empirical performance, it is also slower. For the remainder of this paper, we refer RANLUX(p) as the "standard" 48-bit generators as described with $p - 24$ random numbers discarded.

1.6. Multiple Recursive Generator (MRG)

MRG is an extension of LCG and it computes a linear combination of the past k random numbers generated by

$$x_n = (\alpha_1 x_{n-1} + \cdots + \alpha_k x_{n-k}) \bmod m, \quad n \geqslant k$$

with an initial non-zero vector $(x_0, \ldots, x_{k-1})$, where m is usually a large prime number. Clearly, when $k = 1$, MRG is reduced to LCG. When $m = 2$, MRG is simply the Tausworthe generator in [29].

Finding the coefficients α_i so that it is efficient while achieving the maximum period of $m^k - 1$ is an important and difficult issue. See, for example, [30–33]. A maximum period MRG is known to have the property of equidistribution up to k dimensions: every t-tuple ($1 \leqslant t \leqslant k$) of integers between 0 and $m - 1$ appears exactly the same number of times (m^{k-t}) over its entire period $m^k - 1$, with the exception of the all-zero tuple which appears one time less ($m^{k-t} - 1$). For details, see [30], Theorem 7.43.

LFG (see Section 1.3) is a special case of MRG and only two nonzero terms whose values are ± 1. Following the previous explanations, small multipliers fail to give sufficient "twisting" on the previous set of numbers. This can explain the poor empirical performances for LFG and SWC. A necessary (but not sufficient) condition for an MRG to have a good lattice structure is that the sum of squares of coefficients, $\sum_{i=1}^{k} \alpha_i^2$, is large (see [34]). Therefore, it is critical to find an MRG with a much larger value of α_i and/or to add more nonzero terms while maintaining efficiency and portability.

In the next section, we first consider a general method to improve the performance without losing too much of its generating efficiency. We then consider a special class of MRGs which is motivated by this general method.

2. Twisting and Combining (TAC) methods

As previously discussed, most RNG's are quite efficient with some deficiencies in the empirical study. While we cannot make them to become "perfect" generators, we can use a general method, called Twisting and Combining (TAC) method, to improve the generators. Specifically, let X_1 and X_2 be two random variables corresponding to random number generators. We can twist and combine these two random number generators by

$$Y = N_1 X_1 + N_2 X_2 \bmod 1,$$

where N_1 and N_2 are some constants. Usually N_1 and N_2 are chosen as integer values unless the number of bits of precision (e.g., 24-bit RANLUX) in X_1 or X_2 are small. In that case, choosing a real constant with some decimal digits can increase the number of bits of the precision. In fact, 48-bit decimation version of RANLUX as considered in [28] belongs to this class with $N_1 = 1$ and $N_2 = 1/2^{24}$. Here, $v = w \bmod 1$ means that $0 \leqslant v < 1$ is the fractional part of w where $w - v$ is an integer multiple of 1. In other words, $v = w - \lfloor w \rfloor$, and $\lfloor w \rfloor$ is the floor function which is the largest integer $\leqslant w$. There are two basic operations of twisting and combining in the general method: (1) the term "twisting" is referring to the operation of multiplying some large constants N_1, N_2 to RNGs and (2) the term "combining" is referring to the operation of adding two RNGs. We next discuss these two operations in a greater detail.

2.1. Twisting

Let X be a random variable representing a random sequence generated by a random number generator over the range $(0, 1)$. Suppose that this random number generator can produce many different random numbers that are "dense" over $(0, 1)$ so that it is reasonable to be approximated by a continuous random variable X. For example, the set of possible values of LCG($B; m$) is $\{1/m, 2/m, \ldots, (m - 1)/m\}$, if the multiplier m is a prime and the multiplier B yield the maximum period $(m - 1)$. The gap between two closest numbers is $1/m$ which is decreasing, as m increases. Consequently, we let X be a continuous random variable over $(0, 1)$ and we can "twist" it by

$$Y = NX \bmod 1.$$

Theoretically, it can be proved that $Y \to U(0, 1)$, as $N \to \infty$, for any continuous random variable X (for more general results, see [35]). Multiplying a random variate with N tends to have the effect of twisting the bits of the random variate. Obviously, we should not choose $N = 2^d$ which will only shift the bits but lose binary digits. As a simple example, assume that X represents the random sequence generated by a simple additive generator $x_n = x_{n-1} + 1 \bmod m$, and $u_n = x_n/m$. The resulting sequence is $\langle 0/m, 1/m, 2/m, \ldots, (m - 1)/m \rangle$. While its distribution is uniform, it is clearly not a *random* sequence. For example, the successive pairs taken from the sequence forms a single straight line and it cannot cover the unit-square $[0, 1] \times [0, 1]$. However, twisting X as $Y = NX \bmod 1$ will yield a (slightly) better random sequence by re-shuffling the numbers as $\langle (i * N/m) \bmod 1,\ i = 1, 2, \ldots, m - 1 \rangle$, if $\gcd(N, m) = 1$. The successive pairs taken from the new sequence are now covered by several equal-distance parallel lines and these pairs have a better coverage over the unit-square $[0, 1] \times [0, 1]$. In a sense, LCG($B; m$) also uses this method by twisting (with $N = B$) its previous number as

$$x_n = Bx_{n-1} \bmod m, \qquad u_n = Bu_{n-1} \bmod 1, \quad \text{and}$$
$$u_n = x_n/m.$$

Consequently, we have the sequence (with x_0 as the initial seed selected) as

$$\langle (x_0 \times B^i/m) \bmod 1,\ i = 1, 2, \ldots, m-1 \rangle.$$

Following such an argument, we can see that a small B (corresponding to a "small twisting") in the LCG$(B; m)$ is not a good choice as a multiplier even if it can yield the maximum period.

In theory, twisting method can transform a bad generator into a good RNG with a large multiplier N. However, we should not choose N too large nor $N = 2^d$ which results significant digits lost. As previously mentioned, the multiplier N can be any real number as well.

2.2. *Combining*

Let X_1 and X_2 be two random variables representing two "reasonably good" RNGs (e.g., R250/R1279). To improve generating efficiency, we can simply choose $N_1 = N_2 = \pm 1$ in the TAC method. Therefore, we have

$$Y = X_1 \pm X_2 \bmod 1.$$

The method of combining is not a new idea. As previously mentioned, SWCW is combining SWC and Weyl's additive generator. A simple suggestion in [36] is to add three different LCGs with different multipliers and then take the fractional part. Through a simple example, they claimed that this procedure "ironed out" the imperfections in the component variates. Theoretical justification can be found in [37]. In particular, it was shown that the distribution of Y will be much closer to the $U(0, 1)$ distribution than those of X_1 and X_2. See [35,37,38] for a further discussion. While the combined generator SWCW performed better than generators mentioned in [3], SWCW had some problems in the Swendsen–Wang algorithm as reported in the same paper. One reason is that Weyl's additive generator is a naive and bad generator as a second component of the combined generator.

2.3. *DX generator*

DX generator, proposed by [32] and later extended by [33], is a system of portable and efficient MRGs of modulus m and order k. DX generators are efficient because all nonzero coefficients α_i of the recurrence are equal. DX generators includes FMRG (Fast MRG), proposed by [39] as a special case. The major advantage of a DX generator is that a single multiplication is needed to compute the recurrence, so the generator would run faster than the general case.

For $k = 1597$ and $m = 2^{31} - 1$, several DX generators have been found in [33], each with the maximum period of 10^{14903}:

$$x_n = B(x_{n-t} + x_{n-533} + x_{n-1065} + x_{n-1597}) \bmod m,$$
$$n \geqslant 1597.$$

To avoid the possibility of obtaining 0 or 1, it was recommended that $u_n = (x_n + 0.5)/m$ (see [32]). When $t = 1$, the multiplier B achieving the maximum period found are (a) $B = 1854$ (smallest), (b) $B = 44875$ ($B < \sqrt{m}$), (c) $B = 512675$ ($B < 2^{19}$), and (d) $B = 1073741362 = 29746 \times 36097$ ($B < 2^{30}$). See the discussion in [33] for its usefulness and for a portable implementation for various B listed. DX generators can be more efficient with a special form of $B = 2^p \pm 2^q$ as proposed by [19] for LCG. When $t = 3$, two such generators found in [33] are: (e) $B = 2^{29} + 2^8 = 536871168$ and (f) $B = 2^{28} + 2^{11} = 268437504$. Here, we denote these generators as DX-1597-(a) to DX-1597-(f). The main motivation behind DX generators is the TAC method and DX shares all the nice properties of a maximum period MRG.

3. Empirical evaluation of generators used in Wolff algorithm

Throughout this section, we perform computer simulations on 16×16 Ising square lattices with periodic boundary condition, for which the exact solution is known theoretically [40]. The exact value of the average energy is $-E = 1.4530648528$ [16], using $K_c = \frac{1}{2}\ln(1+\sqrt{2})$. The program of Wolff algorithm used is that from [41], and the program of RNG's used is from [42]. Errors in the simulation can be divided into two parts. One is the error of the algorithm, and the other is the error from the RNG's, which is called the systematic error [40]. Thus, we use σ—the standard deviation of the algorithm [2], to measure the error between the exact value and the simulation result. When the errors are beyond $\pm\, 3\sigma$, the "hidden error" occurs.

We perform an extensive study which is directly comparable to that in [3] in which all the parameters and procedures are specified. We report the simulation results among different RNGs to be specified for possible existence of the "hidden errors". In which case, it will be highlighted in boldface in the table given.

3.1. *Twisting and combining R250 and R1279*

We start with two classical generators R250 and R1279. Let u_n and v_n be two random sequences generated from R250 and R1279. Since R250/R1279 shows hidden errors in [3], we consider the following

$$y_n = N_1 u_n + N_2 v_n \bmod 1,$$

where N_1 and N_2 are some integer values. With various selections of (N_1, N_2), we study the possibility of hidden error for the Wolff algorithm. The results are reported in Table 1.

The first two entries of Table 1 are $(N_1, N_2) = (1, 0)$ and $(N_1, N_2) = (0, 1)$, corresponding to R250 and R1279, respectively. Indeed, there are hidden errors found for R250 and R1279 which are about 43.04σ and 4.28σ. For other entries in Table 1, there is no hidden error found even when for a small case of $(N_1, N_2) = (1, 1)$. This empirical study shows that we do not need "large twisting" and we need a simple "combining" for generators like R250 and R1279.

3.2. *Twisting and combining clearly bad generators*

To demonstrate the effect of twisting, we consider the following "generator":

Table 1
Simulation results for improving R250 and R1279

N_1	N_2	Errors	σ	Error/σ
1	0	0.0009495	0.0000220622	**43.04**
0	1	0.0000577	0.0000134936	**4.28**
1	1	0.0000164	0.0000151623	1.08
5	7	0.0000090	0.0000086642	1.04
9	168	0.0000165	0.0000101874	1.62
12	90	0.0000001	0.0000117046	0.01
38	144	0.0000200	0.0000119245	1.68
55	225	0.0000085	0.0000096546	0.89
56	91	−0.0000009	0.0000077624	−0.11
70	7	−0.0000010	0.0000085224	−0.11
75	167	−0.0000061	0.0000083362	−0.73
77	23	−0.0000030	0.0000108031	−0.27
95	180	0.0000042	0.0000141894	0.30
121	255	−0.0000045	0.0000087933	−0.51
122	255	0.0000010	0.0000077844	0.13
123	255	0.0000085	0.0000127722	0.67
124	255	−0.0000049	0.0000137875	−0.35
125	255	0.0000048	0.0000088784	0.55
126	255	0.0000077	0.0000092956	0.83
127	189	−0.0000151	0.0000136784	−1.10
127	250	−0.0000053	0.0000094129	−0.56
127	251	−0.0000019	0.0000098577	−0.19
127	252	−0.0000050	0.0000137193	−0.36
127	253	0.0000094	0.0000075659	1.25
127	254	−0.0000210	0.0000111683	−1.88
127	255	−0.0000067	0.0000104214	−0.64
173	138	0.0000111	0.0000072226	1.54
225	140	0.0000123	0.0000060813	2.03
243	236	−0.0000096	0.0000092321	−1.03
246	158	0.0000066	0.0000114686	0.58

$$y_n = N_1 u_n^{c_1} + N_2 v_n^{c_2} \bmod 1,$$

where N_1 and N_2 are some integer values, u_n and v_n are two random sequences generated from R250 and R1279, and c_1, c_2 are some constants close to 1, say, $c_1 = 1.1$, and $c_2 = 0.9$. Our empirical study is displayed in Table 2 with various values of (N_1, N_2). The first entry in Table 2 is corresponding to a simple combining with $(N_1, N_2) = (1, 1)$ and it has a (large) hidden error of -781.97σ. For the next two entries, it is corresponding to simple (moderate) "twisting" and no combining with $(N_1, N_2) = (97, 0)$ and $(0, 101)$, respectively. We can still observe the large hidden errors of 585.29σ and -465.36σ. For the fourth entry, we have $(N_1, N_2) = (11, 13)$ the hidden error of -10.13σ is smaller but it is still significant. For the remaining entries, the empirical study shows that TAC is able to restore two (clearly) bad RNGs into a good one, if N_1 and N_2 are large enough.

3.3. *Combining two classic efficient generators*

As previously shown, a simple combining method can improve two classical random number generators such as R250 and R1279. To demonstrate the effect of combining other classical efficient RNGs, one consider following generator:

$$y_n = u_n \pm v_n \bmod 1,$$

where u_n and v_n are two random sequences generated from two RNGs.

Table 2
Simulation results for twisting and combining two bad generators

N_1	N_2	Errors	σ	Error/σ
1	1	−0.0120025	0.0000153491	**−781.97**
97	0	0.0095602	0.0000163341	**585.29**
0	101	−0.0068675	0.0000147573	**−465.36**
11	13	−0.0002195	0.0000216664	**−10.13**
10	248	−0.0000201	0.0000111348	−1.80
32	86	−0.0000225	0.0000128647	−1.75
47	53	−0.0000315	0.0000180521	−1.74
52	198	−0.0000152	0.0000094796	−1.60
76	132	−0.0000044	0.0000056220	−0.77
80	180	0.0000063	0.0000136589	0.46
129	60	−0.0000204	0.0000087492	−2.33
155	122	0.0000072	0.0000128321	0.56
157	23	−0.0000097	0.0000080797	−1.19
169	40	−0.0000068	0.0000120428	−0.56
188	126	−0.0000043	0.0000092370	−0.46
231	175	0.0000166	0.0000150885	1.10
243	208	−0.0000114	0.0000113940	−1.00
250	191	0.0000025	0.0000069747	0.37

Table 3
Simulation results for combining two classical generators

1st RNG	2nd RNG	Errors	σ	Error/σ
R250	LCG-(a)	−0.0000099	0.0000159007	−0.62
R250	LCG-(b)	0.0000011	0.0000133879	0.09
R250	SWC-(a)	−0.0000142	0.0000177368	−0.80
R250	SWC-(b)	0.0000113	0.0000106058	1.07
R1279	LCG-(a)	−0.0000099	0.0000159007	−0.62
R1279	LCG-(b)	−0.0000021	0.0000211056	−0.10
R1279	SWC-(a)	−0.0000086	0.0000248301	−0.34
R1279	SWC-(b)	−0.0000200	0.0000242811	−0.82
SWC-(a)	LCG-(a)	−0.0000299	0.0000105757	−2.82
SWC-(a)	LCG-(b)	0.0000064	0.0000223243	0.29
SWC-(a)	SWC-(b)	0.0000394	0.0000222636	1.77
SWC-(b)	LCG-(a)	0.0000005	0.0000132399	0.04
SWC-(b)	LCG-(b)	0.0000178	0.0000130612	1.37

1. LCG: We consider the following two popular LCGs:
   (a) LCG(16807; $2^{31} - 1$), and
   (b) LCG($2^{15} - 2^{10}$; $2^{31} - 1$);
2. GFSR: We consider two most popular ones:
   (a) R250, and
   (b) R1279;
3. SWC: The following two SWC generators are popular used:
   (a) SWC(24, 10; 2^{24}), and
   (b) SWC(43, 22; $2^{32} - 5$).

We refer the LCGs mentioned above as LCG-(a) for LCG(16807; $2^{31} - 1$) and LCG-(b) for LCG($2^{15} - 2^{10}$; $2^{31} - 1$). Similarly, we refer the SWCs mentioned above as SWC-(a) for SWC(24, 10; 2^{24}) and SWC-(b) for SWC(43, 22; $2^{32} - 5$). Our empirical study is summarized in Table 3. We can see that a simple combining without twisting will be able to restore two efficient RNGs (with some defects of hidden errors) into a better RNG.

Table 4
Simulation results for DX-1597 generators

DX-generator	B	Errors	σ	Error/σ
DX-1597-(a)	1854	−0.0000216	0.00001893219	−1.14
DX-1597-(b)	44875	0.0000381	0.00002565659	1.49
DX-1597-(c)	512675	−0.0000029	0.00001416327	−0.20
DX-1597-(d)	1073741362	−0.0000227	0.00002069314	−1.09
DX-1597-(e)	$2^{29} + 2^8 = 536871168$	−0.0000131	0.00001716261	−0.76
DX-1597-(f)	$2^{28} + 2^{11} = 268437504$	−0.0000200	0.00001643956	−1.21

Table 5
Test results of big crush (160 p-values) and time to generate 10^8 random numbers

RNGs	Time	p-value					
		$> 1 - 10^{-15}$	$> 1 - 10^{-4}$	$> 1 - 10^{-3}$	$< 10^{-3}$	$< 10^{-4}$	$< 10^{-15}$
SWC(24, 10, 2^{24})	1.71 s	8	8	9	71	71	66
SWC(43, 22, $2^{32} - 5$)	2.31 s	0	0	0	22	21	17
RANLUX(389)	51.98 s	–	–	–	–	–	–
LCG(16807, $2^{31} - 1$)	3.72 s	14	18	18	45	44	41
LCG($2^{15} - 2^{10}$, $2^{31} - 1$)	1.27 s	–	–	–	–	–	–
R250	1.49 s	2	3	3	15	15	11
R1279	1.50 s	2	2	2	3	3	3
TAC($127 * \text{R250}$)	4.25 s	1	1	1	5	5	5
TAC($1023 * \text{R250}$)	4.24 s	1	1	1	5	5	4
TAC($1023 * \text{R1279}$)	4.24 s	0	0	1	2	1	1
TAC($127 * \text{R250} + 1023 * \text{R1279}$)	5.95 s	0	0	0	0	0	0
TAC($\text{R250} + \text{R1279}$)	4.79 s	0	0	0	0	0	0
DX-1597-(a)	4.13 s	0	0	0	0	0	0
DX-1597-(b)	4.14 s	0	0	0	0	0	0
DX-1597-(c)	4.11 s	0	0	0	0	0	0
DX-1597-(d)	4.11 s	0	0	0	0	0	0
DX-1597-(e) ($B = 2^{29} + 2^8$)	2.05 s	0	0	0	0	0	0
DX-1597-(f) ($B = 2^{28} + 2^{11}$)	2.04 s	0	0	0	1	0	0

3.4. Using DX generators

Finally, we replace the existing RNG with some of the DX-1597 generators discussed earlier for the Wolff algorithm. The C-program implementation of the DX generator is given in http://www.cs.memphis.edu/~dengl/dx-rng/. Table 4 is the empirical results of Wolff's algorithm where DX-1597 generators are used. As we can see in Table 4, no hidden errors are found in the empirical study when DX generators are used accompanying the Wolff algorithm.

4. Statistical tests and timing comparison

In addition to test these RNGs used in the Wolff algorithm simulations, we use the most extensive and popular RNG package, TestU01 [42], to perform statistical tests on these RNGs. As suggested in the documentation of the above test package [28], we first perform a Small Crush Test; and if necessary, we next proceed to a more extensive Crush Test; and finally to the most extensive Big Crush Test. For each RNG tested, there are 15, 144 and 160 p-values reported by Small Crush Test, Crush Test, and Big Crush Test, respectively.

The test results are summarized in Table 5. The results of tests using the Big Crush Test in TestU01 are tabulated by counting the number of theses p-values which are either too close to 0 or 1. For instance, numbers in the column "$> 1 - 10^{-15}$" and "$< 10^{-3}$" mean the number of p-values which are in the interval $(1 - 10^{-15}, 1]$ and $[0, 10^{-3})$, respectively. A "–" means the Big Crush Test did not apply because either the RNG takes too long (more than one week) to test (like RANLUX) or the RNG has already failed decisively in a Small Crush Test (like LCG($2^{15} - 2^{10}$, $2^{31} - 1$)). The column under the label "Time" gives the time of RNG to generate 10^8 random numbers.

4.1. Timing comparison

In terms of the generating speed, as can be seen from the "Time" column in Table 5, LCG($2^{15} - 2^{10}$, $2^{31} - 1$) is most efficient, followed closely by R250, R1279, SWC, DX-1597(e) and DX-1597(f) generators. The next group of generators are: LCG(16807, $2^{31} - 1$), general DX generators, and TAC generators. RANLUX is clearly several times slower than other generators studied. The timings for generating 10^8 RANLUX(p) random numbers are: 9.95 seconds ($p = 48$), 16.09 seconds ($p = 97$), 31.56 seconds ($p = 223$), and 51.98 seconds ($p = 389$). Using the LUX level recommended in [10], Table 5 shows that the timing of RANLUX(389) is 40+ times of Wu's LCG, 35 times of R250 and R1279, 20 to 30 times of SWC, 9 to 12 times of TAC, and 13 to 25 times of DX generators.

Note that the generating speed is highly hardware and software dependent. The timing results given in Table 5 are meant to serve as a reference guide. Also note that we use the built-in program provided in the TestU01 package for a generating speed comparison. We implement a special algorithm for some specific multipliers of the form $2^p \pm 2^q$ for generators like Wu's LCG$(2^{15} - 2^{10}, 2^{31} - 1)$, DX-1597(e) and DX-1597(f) with $B = 2^{29} + 2^8$ and $B = 2^{28} + 2^{11}$, respectively. As explained in [19], one can use some fast logical operations to replace multiplication and modulus operations for multipliers of the form $2^p \pm 2^q$ and modulus $2^{31} - 1$.

While the generating speed is important, there are many important factors should also be considered. One of such factor is the empirical performance comparison which is discussed next.

4.2. Empirical performance comparison

The results in Table 5 also indicate the poor empirical performances of classical generators such as LCG, SWC, R250, and R1279. In particular, Wu's LCG$(2^{15} - 2^{10}, 2^{31} - 1)$ did not pass the simple Small Crush Test. On the other hand, TAC generators and DX generators have a much better performances than these classical generators. In total, there are $6 \times 160 = 960$ p-values produced for performing Big Crush Tests on 6 DX-1597 generators. Out of these 960 p-values, there is only one p-value which is smaller than 10^{-3}. Similarly, both TAC$(127 * \text{R250} + 1023 * \text{R1279})$ and TAC$(\text{R250} + \text{R1279})$ appear to pass the big crush test. According to this study, "twisting" or "stretching" alone, e.g., TAC$(127 * \text{R250})$, may not be sufficient to greatly improve the classical generators. With the highest luxury level for the 48-bit version of RANLUX as considered and reported in [28] performs quite well. Hence, it does not require additional twisting or combining. The empirical evaluation of RANLUX has been reported in [28], and thus is omitted in Table 5.

DX generators, motivated by the TAC method, preform much better than any other RNGs listed in Table 5. Therefore, DX generators are recommended because of their proven (extreme) long period, their generating efficiency, and their great empirical performance.

5. Summary and conclusion

Either using twisting and combining method on some classical efficient generators or using the DX generators, one can improve the empirical performance of generators used in the Wolff algorithm. No hidden errors are found in the empirical study when "good" generators are used accompanying the Wolff algorithm. From theoretical and practical considerations, we have shown that the TAC method can indeed improve performance of classical RNGs, including those with poor empirical performances. Furthermore, this method give us a new and helpful view of point to generate high quality random numbers rather than to create new RNGs. In short, using TAC method can make existing generators more reliable.

The application to the popular MT19937 can be used as a demonstration of the proposed TAC method. MT19937 proposed by [43] has a period of $2^{19937} - 1 \approx 10^{6001.6}$ and equidistribution property up to 623 dimensions. Because of its long period length, its generating efficiency, and its high-dimensional equidistribution property, MT19937 has become quite popular. However, it was first reported in [28] that MT19937 and its related family of generators failed linear complexity tests, with p-values larger than $1 - 10^{-15}$. Using the TAC method discussed in this paper, one can consider combining with other RNG to improve the empirical performance of MT19937. A simple and efficient method, due to [44], to improve the $U(0, 1)$ sequence x_i, generated by MT19937 can be obtained:

$$u_i = x_{2i} + x_{2i+1} \bmod 1, \quad i = 0, 1, 2, \ldots.$$

Extensive Crush and Big Crush tests on the "combined" MT19937 showed that the improved generator passed battery of tests in TestU01 including linear complexity tests. Detailed discussion and empirical results can be found in [44].

Acknowledgements

The authors are grateful to the Editor and two anonymous referees who made many helpful comments and suggestions that led to a significant improvement of this paper.

References

[1] U. Wolff, Phys. Rev. Lett. 62 (1989) 361.
[2] U. Wolff, Phys. Lett. B 228 (1989) 379.
[3] A.M. Ferrenberg, D.P. Landau, Y.J. Wong, Phys. Rev. Lett. 69 (1992) 3382.
[4] P. Grassberger, Phys. Lett. A 181 (1993) 43.
[5] P.D. Coddington, Int. J. Modern Phys. C 5 (1994) 547.
[6] F. Schmid, N. Wilding, Int. J. Modern Phys. C 6 (1995) 781.
[7] L.N. Shchur, J. Heringa, H. Blöte, Physica A 241 (1997) 579.
[8] L.N. Shchur, H. Blöte, Phys. Rev. E 55 (1997) 4905.
[9] G. Ossola, A.D. Sokal, Phys. Rev. E 70 (2004) 027701.
[10] M. Lüscher, Comput. Phys. Comm. 79 (1994) 100.
[11] M.F. James, Comput. Phys. Comm. 79 (1994) 111.
[12] L.N. Shchur, P. Butera, Int. J. Modern Phys. C 9 (1998) 607.
[13] I. Vattulainen, T. Ala-Nissila, K. Kankaala, Phys. Rev. Lett. 73 (1994) 2513.
[14] I. Vattulainen, T. Ala-Nissila, K. Kankaala, Phys. Rev. E 52 (1995) 3205.
[15] S. Mertens, H. Bauke, Phys. Rev. E 69 (2004) 055702(R).
[16] P.D. Beale, Phys. Rev. Lett. 76 (1996) 78.
[17] W. Janke, Quantum simulations of complex many-body systems: From theory to algorithms, NIC Series 10 (2002) 447.
[18] D.H. Lehmer, in: Proceedings of the Second Symposium on Large Scale Digital Computing Machinery, Harvard University Press, Cambridge, 1951, p. 141.
[19] P.C. Wu, ACM Trans. Model. Comput. Simul. 23 (1997) 255.
[20] P. L'Ecuyer, R. Simard, ACM Trans. Math. Softw. 25 (1999) 367.
[21] T.G. Lewis, W.H. Payne, J. ACM 20 (1973) 456.
[22] S. Kirkpatrick, E.P. Stoll, J. Comput. Phys. 40 (1981) 517.
[23] G. Marsaglia, B. Narasimhan, A. Zaman, Comput. Phys. Comm. 60 (1990) 345.
[24] G. Marsaglia, A. Zaman, Stat. Prob. Lett. 8 (1990) 329.
[25] G. Marsaglia, A. Zaman, Ann. Appl. Prob. 1 (1991) 462.
[26] G. Marsaglia, Proc. Sympos. Appl. Math. 46 (1991) 73.
[27] R. Crandall, C. Pomerance, Prime Numbers—A Computational Perspective, Springer-Verlag, New York, 2000.
[28] P. L'Ecuyer, R. Simard, ACM Trans. Math. Softw. 33 (2007), Article 22.
[29] R.C. Tausworthe, Math. Comp. 19 (1965) 201.
[30] R. Lidl, H. Niederreiter, Introduction to Finite Fields and their Applications, revised ed., Cambridge University Press, Cambridge, 1994.

[31] D.E. Knuth, The Art of Computer Programming, vol. 2, third ed., Addison-Wesley, Reading, MA, 1998.
[32] L.Y. Deng, H.Q. Xu, ACM Trans. Model. Comput. Simul. 13 (2003) 299.
[33] L.Y. Deng, ACM Trans. Model. Comput. Simul. 15 (2005) 1.
[34] P. L'Ecuyer, INFORMS J. Comput. 9 (1997) 57.
[35] L.Y. Deng, D.K.J. Lin, J. Wang, Y. Yuan, Statist. Sinica 7 (1997) 993.
[36] B.A. Wichmann, I.D. Hill, Appl. Statist. 31 (1982) 188.
[37] L.Y. Deng, E.O. George, Comm. in Statist. B 19 (1990) 145.
[38] J.E. Gentle, Random Number Generation and Monte Carlo Methods, second ed., Springer-Verlag, New York, 2003.
[39] L.Y. Deng, D.K.J. Lin, Amer. Statist. 54 (2000) 145.
[40] A.E. Ferdinand, M.E. Fisher, Phys. Rev. 185 (1969) 832.
[41] J.S. Wang, Wolff.c, http://www.cz3.nus.edu.sg/~wangjs/BeijingWorkshop.html, 2002.
[42] P. L'Ecuyer, R. Simard, TestU01: Empirical testing of random number generators, http://www.iro.umontreal.ca/~simardr/testu01/tu01.html, 2006.
[43] M. Matsumoto, T. Nishimura, ACM Trans. Model. Comput. Simul. 8 (1998) 3.
[44] L.Y. Deng, H.H.S. Lu, T.B. Chen, 64-bit and 128-bit DX random number generators, Preprint, 2007.