Journal of Universal Computer Science, vol. 12, no. 6 (2006), 672-690
submitted: 31/5/06, accepted: 23/6/06, appeared: 28/6/06 © J.UCS

Pseudorandom Number Generation: Impossibility and Compromise

Makoto Matsumoto,[1] **Mutsuo Saito,**[2] **Hiroshi Haramoto,**[3] **Takuji Nishimura**[4]
([1,2,3] Department of Mathematics, Hiroshima University, Japan
[4] Department of Mathematics, Yamagata University, Japan
m-mat@math.sci.hiroshima-u.ac.jp)

Abstract: Pseudorandom number generators are widely used in the area of simulation. Defective generators are still widely used in standard library programs, although better pseudorandom number generators such as the Mersenne Twister are freely available.

This manuscript gives a brief explanation on pseudorandom number generators for Monte Carlo simulation. The existing definitions of pseudorandomness are not satisfactorially practical, since the generation of sequences satisfying the definitions is sometimes impossible, somtimes rather slow. As a compromise, to design a fast and reliable generator, some mathematical indices are used as measures of pseudorandomness, such as the period and the higher-dimensional equidistribution property. There is no rigorous justification for the use of these indices as measures of pseudorandomness, but experiences show their usefulness in choosing pseudorandom number generators.

Key Words: Random number generation, Pseudorandom number generation, Mersenne Twister, Monte Carlo methods, Simulation

Category: G.3 Mathematics of Computing, Probability and Statistics, Random number generation

1 Introduction

The authors of this manuscript take the view that there exists no commonly accepted perspective in the research of pseudorandom number generation. Actually, each group of researchers have their own dogma, which is sometimes contradicting those of the others. This manuscript is intended as an unbiased survey, but still should suffer from our own biases. A clear bias is that one aim of this manuscript is to advertise a pseudorandom number generator "Mersenne Twister" (MT) [Matsumoto and Nishimura 1998]. This generator has the period of length $2^{19937} - 1$, has the 623-dimensional equidistribution property, and can generate more than 10^7 pseudorandom 32-bit integers per second in standard desk-top computers. It is now widely used: implemented in various computer languages by the hands of volunteers, and downloadable as free software.

As we shall mention in §2.2, the most serious problem surrounding pseudorandom number generators (PRNGs) seems the long distance between the users and the developers. There are a lot of defective generators adopted in common libraries still now, although many excellent generators are freely downloadable. We wish to get the users and the developers closer, which is another aim of this manuscript.

2 Defective PRNGs and Mersenne Twister

Many defective PRNGs are currently being used. These are not necessarily old: some of the newly proposed generators are still defective. Here is an example. The third edition of Knuth's famous book [Knuth 1997] was published in 1997. This series of books is known as "the bible of computer science," and many researchers (especially non-specialists in random number generation) refer to the book as the best reliable source of PRNGs.

However, most generators there were developed in the 1970's, and show some defect in large scale simulations. The third edtion newly introduced a generator named `ran_array`. This is a modification of a classical defective generator, the "lagged-Fibonacci generator," improved by discarding a large part of the output sequence. This modification seems not smart: the improved generator still has an observable deviation [Matsumoto and Nishimura 2003], if the luxury level is 3 or less (i.e., if less than 90% is discarded.)

The first author met Knuth in Japan in '96. A draft of the third edition was downloadable, and the first author (too young) claimed to Knuth: "It is a serious issue that such an influential book, called the bible, does not treat good PRNGs." Knuth's reply was matured: "Those who want to call this book the bible should call it the Old Testament. There are more urgent problems in this series which I need to address, so I cannot pay enough attention to the part of PRNG. . . . Your PRNG seems interesting, but not well-tested. If it is widely used and survives, then it may be introduced in the fourth edition, planned fifteen years later." He then gave the first author a number of valuable comments, including a suggestion on the name of Mersenne Twister. We also think Knuth's book is worth being called the bible.

We show two examples of classical generators, still being widely used but giving erroneous results.

2.1 Linear Congruential Generator

The Linear Congruential Generator (LCG) has been the most widely used, standard PRNG (Lehmer, introduced around 1960). Let a, c, N be integer constants. LCG is to generate a sequence of integers $x_0, x_1, \ldots$ between 0 and $N-1$ by the recursion

$$x_{j+1} := ax_j + c \mod N \quad (j = 0, 1, 2, \ldots), \tag{1}$$

and use them as a pseudorandom integer sequence. Here mod N denotes the residue modulo N. The user needs to specify x_0 as the initial seed. If one needs another sequence, then one changes the initial seed. Once we have $x_j = x_{j+p}$ for some j and $p > 0$, then this holds for any larger j. Since there are at most N possible values for x_j, the period of this sequence is at most N.

Often the user wants uniform random real numbers in the half-open interval [0,1). They are usually obtained by the conversion $x_j \mapsto x_j/N \in [0, 1)$.

Since N gives the upper bound on the period and the resolution, a large N is desirable. On the other hand, we need to compute integer multiplications modulo N, which is expensive if N is large, in particular if N exceeds the word size of the CPU. Consequently, most of the implemented LCGs adopt $N = 2^{32}$ or $N = 2^{48}$. Such choices might have been enough 20 years ago, but recent personal computers can generate 2^{32} integers in a few minutes by such an LCG. In a nuclear physics simulation, even 2^{40} random numbers are consumed. So, the periods of such LCGs are too short now.

Another problem of LCGs is the lattice structure appearing in high-dimensional random points plotting. Here we choose an implementation of the `rand` random number generating function, included as an example in an ANSI document of C language in the middle 80s. It is an LCG with parameters $a = 1103515245$, $c = 12345$, $N = 2^{31}$ in the notation of §2.1. Generate random points in the 3-dimensional unit cube by this LCG as follows.

1. Generate three pseudorandom numbers in the unit interval $[0, 1)$, and plot a point at the corresponding coordinate in the unit cube.
2. Iterate 1, for 2^{31} times.
3. Cut out a small cube of size 0.015^3 from the unit cube, and draw the picture of the plotted points inside.

Then we obtain Figure 1. Comparing with Figure 2 generated by MT, one sees a clear lattice structure. This is not because of bad choices of a or c. For any LCG, if it is used for the full period, then the generated points in the unit cube are arranged in a lattice. (The above value of a is well-chosen so that the lattice unit is close to a cube, rather than a thin parallelepipe, which shows a better randomness. See [Knuth 1997, §3.3.4].)

Another warning applies when the modulus N is chosen to be a power of two, to decrease the cost of calculating modulo N. In this case, the s least significant bits of the generated pseudorandom integers have the period at most 2^s. In particular, the least significant bit has period at most 2, and the above LCG generates odd numbers and even numbers alternately.

In ANSI-C, there is only a specification of `rand` (i.e., the range of the numbers generated), and no algorithm is specified. The above LCG is listed as an

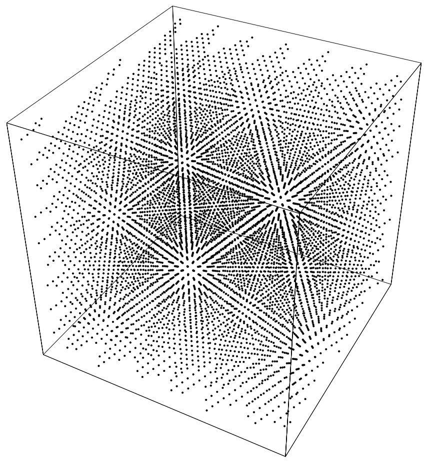
Figure 1: "Random" points in a cube, generated by a standard LCG `rand`.

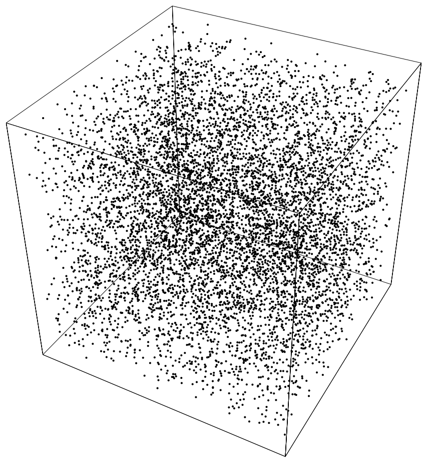
Figure 2: "Random" points generated by Mersenne Twister.

example of implementation, and had been often the default until the mid 1980's. Presently, `rand` in the BSD Standard C Library is replaced with a lagged Fibonacci generator (named `random` in the library), but this latter generator too has serious deviations [Matsumoto and Nishimura 2003, Matsumoto et al. 2006].

2.2 GFSR Generators

The Generalized Feedback Shift Register (GFSR, [Lewis and Payne 1973]) is another example of a widely used but defective generator.

Assume that a CPU uses w-bit words, and then one word can be regarded as a w-dimensional horizontal vectors with coefficients in $\{0, 1\}$. A (three-term) GFSR is to generate a pseudorandom word sequence $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots$ by the recursion

$$\mathbf{x}_{j+n} := \mathbf{x}_{j+m} \oplus \mathbf{x}_j \quad (j = 0, 1, \ldots), \tag{2}$$

where $\oplus$ denotes the component-wise addition modulo two (i.e., bitwise exclusive-or in the computer terminology, or the addition of vectors with components in the two element field $\mathbb{F}_2 = \{0, 1\}$).

The first n words $\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}$ should be given as the initial values by the user. If the integers $n > m > 0$ are chosen so that $t^n + t^m + 1$ is a "primitive polynomial over $\mathbb{F}_2$" (see for example [Niederreiter 1992] for a definition), then the period of this sequence attains the upper bound $2^n - 1$. In implementations, values n with $60 \leq n \leq 1000$ are often used. In the implementation, one needs to keep the last n outputs to compute the recursion, so a memory of n words is necessary.

GFSR has a large period, and requires only a few CPU instructions to generate one word. Consequently, it has been used in large scale applications, but the balance between the number of 0's and 1's is known to deviate. Fix any bit in the words, say, the most significant bit (MSB). Fix an integer N. Take every N-tuple of the output words, and count the number of 1's in the MSBs of the N-words. If the words are random, then it should conform to the binomial distribution $B(N, 1/2)$, but the three-term GFSR has a huge deviation for $N > n$ (no deviation is observed for $N \leq n$). Roughly speaking, this is because the present output bit is the exclusive-or of only two bits in the past n bits, so if there are many 0's (or many 1s) in the past N bits, then the next bit tends to be 0 more than 1. This deviation sometimes leads to erroneous results in a simulation of a Markov process. Random walks and Ising models are typical examples. In such simulations, the state of the model is influenced by all the past outputs, so the deviation of n-tuples tends to lead errors in sumulation.

The *weight distribution test* is to measure the deviation from $B(N, 1/2)$ by using a standard chi-square test (the same test is called the frequency test in the NIST randomness test package). Three term GFSRs are known to be rejected by this test (e.g., the result of tests in [Matsumoto and Kurita 1992]). This fact had been pointed out and analyzed by [Lindholm 1968] and [Fredricsson 1975], but the users (and even some designers) of PRNGs have not paid enough attention.

[Ferrenberg et al. 1992] reported that such PRNGs yield erroneous values of the phase transition temperature in Ising model simulations (without analyzing the reasons), and attracted people's attention. However, the defect itself had been known 24 years before.

The serious problem in researching PRNGs seems a social problem: PRNGs, which are known to be defective, are still widely used, and are even newly implemented.

Although reliable fast PRNGs exist, they tend to be neglected because the theory behind them is not simple. Because of the lack of a practical definition of pseudorandomness, only the "dogmas" of authorities are reflected in the implementations. Classical PRNGs (known to be defective to specialists) and their slightly improved versions are repeatedly recommended and used, then some of the defects are (re)discovered by the users. Then, minor modification is added to shrink the defects. A few years later, the advance of the performance of computers again reveals that the remaining defects, although reduced, persist. As a result, many defective generators have been standardly used and advertised, neglecting the high-performance PRNGs that have already existed.

Because of these reasons, not only a few users of PRNGs seem to consider any PRNG as a non-reliable source of pseudorandomness. We sometimes see comments such as: “If one uses more than 10^7 random numbers, then one should use physically generated ones from noise, rather than PRNGs.” We disagree with such comments, since modern generators such as MT pass stringent tests on randomness with more than 10^9 samples, and give satisfactorial results in large scale simulations.

By extending the ideas in [Fredricsson 1975], we developed an algorithm to compute the deviation of PRNGs (with linear recursion over $\mathbb{F}_2$) from the binomial distribution, using the MacWilliams identity from Coding Theory. We call this test *weight discrepancy test* [Matsumoto and Nishimura 2002]. This test determines beyond which sample size the PRNG will be rejected by the weight distribution test. For example, it claims that a three-term GFSR (2) with $n = 89$ will be rejected for more than 10^5 samples, while a similar GFSR with $n = 521$ will be rejected for more than 10^7 samples. On the other hand, a toy model of MT with 521 bits of internal state requires 10^{156} samples to be rejected by the weight distribution test. Such conclusions cannot be deduced by empirical statistical tests. There are many statistical tests on the randomness of PRNGs. According to our experiences, the weight distribution/discrepancy test is one of the strongest tests to detect the failure of $\mathbb{F}_2$-linear PRNGs.

So far, we saw concrete problems in PRNGs. In the next section, we shall very briefly survey the methods of random number generation.

3 Generality: Pseudorandom Number Generation

3.1 Random Numbers

Random numbers have been used since ancient times, e.g., a dice is an example of a random number generator. However, generating “high-quality” random numbers by hand is not so easy. L.H.C. Tippett tries to generate random numbers by choosing a card randomly from a set of numbered cards by hand. He found that the result was strongly deviated, and noticed the difficulty in generating randomness by hand. Then he made the first published random number table in the world, since he considered it worth being published.

By the development of computers, random numbers are used more widely and extensively now. Simulations of probabilistic events requires random numbers. Such a use of random numbers, or any related numerical computations based on random numbers, are called Monte Carlo methods.

The questions in the application are

– how to generate (pseudo) random numbers, and

– how to assure their randomness (at least some safety for the purpose).

Unfortunately, the situation is “there are no complete answers” presently (and probably no even in the future.) One of the reasons is the lack of enough practical and reasonable definition of pseudorandomness at present. Actually there are definitions, but generating sequences satisfying them is difficult and takes CPU-time (as will be explained in §3.5).

3.2 Naive Definition

Definition 1. Let $X_1, X_2, \ldots$ be independent random variables conforming to an identical distribution. A random number sequence is a series of realized values $x_1, x_2, \ldots$ of these random variables.

This is the naive idea, but there is no mathematical definition of "a realized value" of a random variable. For example, consider an ideal dice. The i-th output is X_i, which takes values $\{1, 2, 3, 4, 5, 6\}$ with probability $1/6$ for each possiblitity, and $\{X_i\}$ are independent. This is a perfect description of the probabilistic situation, but gives no way to generate random numbers.

The definition of pseudorandom number generators (PRNGs) are even more ambiguous:

Definition 2. A pseudorandom number generator is an algorithm generating a sequence of numbers that appears to be random, by some deterministic algorithm.

3.3 Physical Random Numbers

We postpone discussions on PRNGs to the next section. A most naive random number generation is to sample a physical noise and digitize it. Such methods are called physical random number generations.

A problem of such generators is the cost for the hardware. It is not easy to detect any malfunction of such devices. It should be independent of the circumstance such as temperature etc. There are commercial devices for this purpose, but most of them are combined with PRNGs to eliminate unbalancedness and dependency on the circumstance.

A more intrinsic problem is that physical random numbers have no reproducibility, i.e., one needs to record all the generated numbers if one would like to reproduce the whole simulation. This is often inconvenient, in particular when a user wants to re-test another user's simulation.

3.4 Pseudorandom Number Generation

A PRNG is to generate a sequence of numbers by a deterministic algorithm (usually expressed by a recurrence formula, with possible output conversion), and to use it as a random number sequence. We have already seen a few examples in the previous sections.

Once the recurrence formula and the initial value are given, anyone can generate the same sequence any times. It is possible to assure some property of the sequence mathematically, such as the length of the period and the distribution, differently from physical random numbers. The problem is that we do not know "which sequence can be used as a random number sequence."

3.5 Definition of Random/Pseudorandom Number Sequences

To say the truth, there are rigorous definitions of randomness. A rough description of the definition by Kolmogorov and Chaitin is: "A finite sequence of numbers is a random number sequence if there is no description of the sequence shorter than the sequence itself," i.e., no computer programs shorter than the sequence can produce the sequence. A lot of interesting mathematical properties (including existence of such sequences) can be proved, but this definition is too restrictive for efficient random number generation.

A less strict but still strong definition, adopted in the area of cryptography, is as follows: "a sequence of numbers is said to be a computationally secure pseudorandom number sequence, if it can be generated by a polynomial-time algorithm (when the recursion and the initial values are given), but there is no polynomial-time algorithm to guess the next outputs. This is a perfect definition, since it is proved that any methods (including statistical tests and simulations) which can be computed in polynomial-time can never distinguish the sequence from a genuine random number sequence.

A problem is whether such a sequence exists or not. Under a strong hypothesis on the hardness of integer factorization, such a generator is proved to exist by [Blum et al. 1986] (called BBS). If there is a polynomial-time algorithm to guess the next output of BBS generators, then this gives a polynomial-time algorithm for (a class of) integer factorizations, which is considered unlikely since mathematicians could not do that for decades.

Such PRNGs are widely used for cryptographic purposes. However they are much slower than the generators we have seen so far. Moreover, mathematical analysis such as the distribution in high dimensions is difficult. Usually, such PRNGs are not used in a large scale Monte Carlo simulation where the generation speed is crucial.

4 Examples of PRNGs

Under the lack of good definitions of pseudorandomness for Monte Carlo simulation, various recurring sequences are proposed and used as PRNGs.

4.1 Automaton

PRNGs can be described as finite state automata.

Definition 3. (Automaton.)

Let S be a finite set (of the possible states of the memory assigned for the PRNG). Let $f : S \to S$ be a function (state transition). Let $s_0 \in S$ be the initial state. Every time unit, the state is changed by the recursion

$$s_0, s_1 := f(s_0), s_2 := f(s_1), \ldots ,$$

and the output sequence is generated as

$$o(s_0), o(s_1), o(s_2), \ldots$$

by applying an output function $o : S \to O$, where O is the set of possible output numbers. This system is called a (no-input finite state) automaton.

As far as the amount of the memory is finite and no input is given, any digital computer is such a system. Since the number of the states is finite, the state transition becomes periodic in the long run, and the period length is bounded by $\#(S)$.

In other words, if a PRNG has no period, then it consumes more and more memory, in accordance with the generation. Such PRNGs are often proposed, but in practice the amount of memory allowed to be occupied by the PRNG is limited. Then, it is usually better to specify the limit in the design stage, and then maximize the period with that memory size.

This is an example of a gap between theory and implementation. Similar gaps often exist in PRNGs based on “probabilistic theory” or “chaos theory.” For example, some generator (claimed to have probabilistically assured randomness) requires a uniformly and randomly chosen real number in the interval $[0, 1]$ with infinite precision as a seed, which is absurd. (If such a choice was possible, then its (say) floating point expansion would give a perfect random number sequence.)

Quite often, PRNGs based on chaotic dynamics are proposed, but most of them do not have the claimed good randomness. Chaos theory is based on a continuum state space S and a (usually piecewise continuous) transition function $f : S \to S$. However, in a digital computer, S is approximated by some finite discrete set. Since the approximation error is magnified by iteration of f, the behavior of the original continuous dynamical system is totally different from the implemented approximated system. As an example, the `ranlux` generator [Lüscher 1994] is designed by using chaos theory, but we observe strong correlations between the output sequences generated from correlated initial seeds [Matsumoto et al. 2006].

Our feeling is that designing a PRNG is more closely related to the area of discrete optimization (i.e., optimize the period and the distribution) rather than to probability theory or chaos theory.

4.2 Linear Recurrence

LCG treated in §2.1 is an automaton (Definition 3) with $S := \mathbb{Z}/N$ (i.e., the set of residues modulo N) and the transition function is given by a linear function

$$f(s) = as + c \mod N.$$

A PRNG is called linear if the transition function is a linear function in some sense. Non-linear transition functions have been proposed, such as quadratic functions or fractional linear transformations. However, the generation speed is usually slower than linear generators. The computation of its period or distribution is often harder.

Actually, the first PRNG on a digital computer introduced by von Neumann in the 1940's adopted a quadratic transition function: for an integer x, $f(x)$ is given as several middle digits of x^2. But this method sometimes yields a short period [Knuth 1997, §3.1]. LCGs were selected through such trials and errors. Intuitively, linear functions seem to be a poor source for pseudorandom numbers. However, it turns out that an LCG is a relatively good PRNG if the parameters are carefully selected. It had been one of the standard generators, but it is now obsolete (§2.1).

4.3 Use of Finite Fields

Let $\mathbb{F}_2 = \{0, 1\}$ be the two element field, i.e., addition and multiplication are done modulo 2.

An automaton (Definition 3) with a state space $S = \mathbb{F}_2^d$ and an $\mathbb{F}_2$-linear transition function $f : \mathbb{F}_2^d \to \mathbb{F}_2^d$ (i.e., a $(d \times d)$-matrix with coefficients in $\{0, 1\}$) is called an $\mathbb{F}_2$-linear PRNG.

The (three-term) GFSR (2) explained in §2.2 lies in this class. The state space S is the nw-dimensional $\mathbb{F}_2$ vector space $S := (\mathbb{F}_2^w)^n$, considered as the set of n-tuples of $\mathbb{F}_2^w$. Transition function is

$$f : (\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) \mapsto (\mathbf{x}_1, \ldots, \mathbf{x}_{d-1}, \mathbf{x}_0 + \mathbf{x}_m).$$

It is easy to see that this yields the sequence defined by the recursion (2). One can prove that the period of this sequence is at most $2^n - 1$. The distribution of the number of 1's in the sequence is deviated, as seen in §2.2.

[Matsumoto and Kurita 1992] proposed to introduce a $(w \times w)$-matrix A into (2):

$$\mathbf{x}_{j+n} := \mathbf{x}_{j+m} \oplus \mathbf{x}_j A \quad (j = 0, 1, \ldots), \tag{3}$$

and named it the twisted GFSR. To realize fast multiplication, A has the form

$$\begin{pmatrix} & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \\ a_0 & a_1 & \cdots & \cdots & a_{w-1} \end{pmatrix}$$

for which we can compute the multiplication by simple bit operations:

$$\mathbf{x}A = \begin{cases} \text{shiftright}(\mathbf{x}) & \text{(if the least significant bit of } \mathbf{x} \text{ is 0)} \\ \text{shiftright}(\mathbf{x}) \oplus \mathbf{a} & \text{(if the least significant bit of } \mathbf{x} \text{ is 1)}, \end{cases}$$

where $\mathbf{a}$ denotes the bottom row vector of A. Introducing A results in mixing information of bits with different significance. The period of the twisted GFSR is at most $2^{nw} - 1$, and many parameters attaining this bound are found. Later [Matsumoto and Kurita 1994] introduced a $(d \times d)$ matrix T (named the tempering matrix) and output $\mathbf{x}T$ as the pseudorandom sequence, to improve the higher-dimensional equidistribution property (see §5.2.2). The implemented C-code is named `tt800`, and has been widely used. A large n is desired for long period and high-dimensional distribution, but the limitation is that we need to know the factorization of $2^{nw} - 1$, which is often difficult for $n > 2000$.

MT is a slight modification of the twisted GFSR, with the recursion

$$\mathbf{x}_{j+n} = \mathbf{x}_{j+m} + (\mathbf{x}_j^{w-r} * {}^r\mathbf{x}_{j+1})A,$$

where $\mathbf{x}_j^{w-r} * {}^r\mathbf{x}_{j+1}$ denotes the upper $w - r$ bits of $\mathbf{x}_j$ concatenated with the lower r bits of $\mathbf{x}_{j+1}$.

Concatenating the upper bits of $\mathbf{x}_j$ and the lower bits of $\mathbf{x}_{j+1}$ yields a w-dimensional vector, so A can be multiplied from the right. This recursion makes the dimension of the state space $nw - r$ (since the least significant r bits of $\mathbf{x}_j$ will not be feedbacked). By choosing r such that $2^{nw-r} - 1$ is a prime (a so-called Mersenne prime), we don't need to factorize it. An implemented C-code `mt19937` has a period of length $2^{19937} - 1$, with parameters $n = 624$, $w = 32$, $r = 31$.

A big advantage of GFSR-type generators over LCGs is that the generation speed is constant independently of the degree of recursion, or in other words, independently of the period length (see [Knuth 1997, P.28 Algorithm A]). Another advantage is easiness in improving higher-dimensional distribution, which will be discussed in §5.2.2.

5 Evaluation of PRNGs

Because of the lack of an adequate definition of pseudorandomness, there is no theory nor are there methods that assure genuine pseudorandomness.

5.1 Statistical Tests

On the other hand, there are many methods to show the non-randomness of (poor) PRNGs. A statistical test is to choose some statistical value x computed from the sequence, such as the number of ones in a bit stream, and test its plausibility. Under the null hypothesis that the sequence is a uniformly and independently distributed random sequence, (at least an approximation of) the distribution of x must be obtained in advance. Let X be a random variable conforming to this distribution. Compute the probability that x or more deviated value is observed under the same hypothesis. For example, compute the probability that

$$|X - E(X)| \geq |x - E(X)|$$

holds, and if this probability is too small, say 0.001, then such a deviated observed value x may occur with only once in 1000 times, so the hypothesis that the sequence is uniformly random is rejected with level of significance 0.001 (see for example [Knuth 1997, §3.3]).

Such statistical tests are applicable to any type of random number generators. It is necessary for PRNGs to pass various statistical tests. However, similarly to the case of “definition” of pseudorandomness, the situation is not clear. Which statistical quantity is a useful index of randomness? There are thousands of statistical quantities, and which should be tested? With which sample size? What are the relations between these tests?

Although these questions are open, there are several practical packages for statistical tests of randomness. Marsaglia’s diehard battery [Marsaglia] is rather old but still widely used. [L’Ecuyer and Simard] proposes newer and more extensive test package `testU01`. MT passes all these tests.

But, what does it mean to pass statistical tests? We often get an email saying “I tested MT by a statistical test, then rejection with significance level 0.05 was observed 5 times among 100 trials. Doesn’t this mean MT’s deviation?” Of course it doesn’t. This is the average for genuine random numbers.

However, when we conduct statistical tests to PRNGs many times, we often feel confusion. Is it suspicious or not, if we observe one time rejection of 0.01 significance level among ten tests?

A possible strategy to avoid this confusion is to take larger and larger sample sizes, if some suspicious phenomenon is observed. If a PRNG has some deviation, and if it is detected by a statistical test with some sample size, then the observed probability value tends to become unusual (such as 10^{-6}) by raising the size of samples. When a statistical test is applied to a PRNG, existence of a few suspicious probability values (such as 0.001) does not necessary imply a defect of the PRNG. To confirm the defect, it is better to increase the sample size until some marvelous values of the probability, such as 10^{-6}, can be observed stably.

Another difficulty in the statistical tests is how to integrate the probability values obtained by iterating the same tests. Often, the Kolgomolov–Smirnof Test is conducted to the set of probability values (called double test, [Knuth 1997, §3.3.1]). However, the probability value is often obtained by an approximative formula, and the double test may accumulate errors. We experienced a false-rejection of good PRNGs because of the approximation error. When one obtains a suspicious result in a statistical test on a PRNG, then it is necessary to test other PRNGs and compare the results, to certify the relevance of the test.

5.2 Theoretical Evaluation

Statistical tests are empirical, and the obtained evaluation is probabilistic and has some uncertainity. There are non-empirical evaluations on PRNGs, sometimes called *theoretical evaluation*.

If a pseudorandom number sequence is generated by some mathematical formula, then the sequence may have some mathematical structures. We define some index on the structure and use it as an index of the evaluation of the PRNG. This is the theoretical evaluation. A concrete example of such indices is the period. The notion of the period does not exist in genuine random sequences. The period is a mathematical structure shared by any PRNG, whose model is a finite state automaton. Obviously, a longer period is desired for pseudorandom number generation. However, the period length does not measure the randomness itself. That measures something which is loosely related to the (undefined) "pseudorandomness."

A problem of theoretical evaluations is that a different type of generator has a different type of mathematical structure. Consequently, one test is applicable only to one type of generators, which makes it difficult to compare different types of generators by such a test.

Another shortcoming is that we have no rigorous justification of the evaluation by the index. As in the case of the period, the mathematical definition of each index is clear, but its meaning with respect to the randomness is not so clear. To understand the evaluation, one needs to understand the corresponding mathematical structure.

On the other hand, theoretical evaluation is much more sensitive than statistical test in detecting the defects. Moreover, we can compare good parameters and choose the best one. This is different from statistical tests, where the test result is probabilistic and does not distinguish good PRNGs.

According to this performance of theoretical evaluation, it is used in choosing the parameters in a recursion formula at the designing stage of PRNGs. After choosing the parameters, the designers confirm the quality of PRNG by various statistical tests.

Note, however, that optimizing the index does not mean optimizing "the randomness." For example, a longer period is better, but a long period sequence is not always a good pseudorandom number sequence.

5.2.1 Period

The most common theoretical evaluation is the period. Most of LCGs have period length around $2^{32} \sim 2^{48}$. The standard generator `random` in the BSD Standard C Library has a period around 2^{63}. The commonly used MT19937 has the period $2^{19937} - 1$.

What is a sufficient period length? There is no clear answer. A thumbnail rule is that the period should be at least the cube of the number of actually consumed pseudorandom numbers.

The period is bounded by the cardinality $\#(S)$ of the state space of the PRNG. By a historical reason, a PRNG is said to have *maximal period* if the period is $\#(S) - 1$.

5.2.2 Higher-Dimensional Equidistribution Property

Another index for a theoretical evaluation is the dimension of the equidistribution. Let us assume that an M-tuple of pseudorandom numbers is consumed in some unit in the simulation. (For example, in the case where a function with M random variables is evaluated many times.) Assume that for each unit, the initial state is randomly chosen. Then, a PRNG based on an automaton (Definition 3) can be considered as a function

$$g : S \to O^M \tag{4}$$

mapping the initial state s_0 to the M-tuple of outputs

$$o(s_0), o(f(s_0)), \ldots, o(f^{M-1}(s_0)).$$

Recall that O is the finite set of possible output numbers.

A PRNG is said to have M-dimensional equidistribution property, if it has the maximal period (§5.2.1) and if the output M-tuples are uniformly distributed over O^M when the initial state s_0 is uniformly randomly chosen from S.

Because of the maximality of the period, every state is realized once over a whole period (except for one exceptional state). Consequently, if we observe all the (overlapping) M-tuples appearing in one period, then every possible pattern in O^M appears equally often (except for one exceptional pattern).

The maximum value of such M is called the dimension of the equidistribution of the PRNG. A large M is desirable. An obvious necessary condition is deduced from the surjectivity of g in (4):

$$\#(O)^M \leq \#(S), \text{ or equivalently } M \leq \lfloor \log \#(S) / \log \#(O) \rfloor.$$

GFSR does not attain this upper bound (i.e., the equality at the right hand side), but the twisted GFSR and MT do: MT19937 is M=623-dimensionally equidistributed with $623 = \lfloor 19937/32 \rfloor$.

As usual for theoretical evaluations, it is not very clear whether this property assures pseudorandomness or not. However, experiences show that a small fraction of the full period of an M-dimensionally equidistributed pseudorandom number sequence passes statistical tests based on a function with M or less (uniformly random) variables.

For example, consider a (one-dimensional) random walk of s steps, simulated by s random bits (thus $O = \{0, 1\}$). The last position is the difference between the number of 1's and that of 0's. Experiments show that if $s \leq M$, then no deviation is (usually) observed. On the other hand, (one bit of) three-term GFSRs (2) show deviations if s is greater than M (it is known that $M = n$ holds for one-bit sequences of output of the GFSR).

We can compute the dimension of the equidistribution by linear algebra, if the PRNG is $\mathbb{F}_2$-linear (i.e., f and o in Definition 3 are both $\mathbb{F}_2$-linear) and has the maximal period. This is because the M-dimensional equidistribution property is equivalent to the surjectivity of g (4) (since g is $\mathbb{F}_2$-linear and the linearity implies that the inverse image of any element has the same cardinality). The surjectivity is reduced to the computation of the rank of g, which can be obtained by Gaussian elimination of the matrix. However, for huge state generators (e.g., 19937-dimension for MT), Gaussian elimination is expensive. We used a much faster algorithm based on the lattice structure over power series ring, introduced by [Couture et al. 1993].

There is a more refined notion of *equidistribution* with v-bit accuracy. Assume that O is the set of w-bit integers, say, 32-bit integers. In a Monte Carlo simulation, these integers are often normalized to real numbers in the $[0, 1)$ interval. In that case, the upper bits are more influential than the lower ones. Reflecting this, the following index of evaluation is often used.

Definition 4. Let $\mathrm{tr}_v : O \to \{0, 1\}^v$ be the function that takes the v upper bits of w-bit integers and truncates the other ones (so tr stands for truncation). If the M-tuples of outputs with v-bit accuracy, namely the M-tuples

$$\mathrm{tr}_v(o(s_0)), \mathrm{tr}_v(o(f(s_0))), \ldots, \mathrm{tr}_v(o(f^{M-1}(s_0)))$$

are equidistributed in the above sense, then the PRNG is said to be M-dimensionally equidistributed with v-bit accuracy, and the maximum value of such an M is *the dimension of equidistribution with v-bit accuracy* and denoted by $k(v)$.

The same argument on the surjectivity implies the following upper bounds:

$$k(v) \leq \lfloor \dim(S)/v \rfloor \quad v = 1, 2, \ldots .$$

If this upper bound is attained for every v, $1 \leq v \leq w$, then the PRNG is said to be optimally equidistributed.

MT19937 is not optimally equidistributed. It satisfies these upper bounds for v in $\{1, 2, 4, 8, 16, 32\}$, but for example for v in $\{3, 5\}$, $k(v)$ is about 93% of the upper bound. It would be possible to modify the output function to attain the optimal equidistribution, but it results in some speed-down. Moreover, it is not very clear whether the optimal equidistribution is absolutely important or not. For example, the $k(3)$ of MT19937 is 6240, and the upper bound is 6645. It seems difficult to distinguish such a difference by any reasonable (non-artificial) statistical test. In some applications, the lower bits are more important than the higher ones. In such applications, the definition of the optimal equidistribution is not appropriate. (In the case of MT, it is confirmed by computation that the lower bits have satisfactorial equidistribution property.)

One may argue that it is better to use the CPU-time for some non-linear output conversions, rather than for realizing optimal distributions.

Anyway, if there is a fast, optimally equidistributed PRNG with long period, it is desirable. The WELL generators [Panneton et al. 2006] are optimally equidistributed, and robust to bad initializations. One of them has a period of $2^{44497} - 1$. They are a little slower than MT.

Here we remark on the most famous theoretical evaluation, namely, the spectral test. This test is for LCGs and their relatives, which corresponds to the higher-dimensional equidistribution property. The test measures the thinness of the lattice unit (cf. Figure 1), and is known as the strongest test for LCGs. We refer to [Knuth 1997, §3.3.4] for detail.

5.2.3 Weight Discrepancy Test

As we mentioned in §2.2, the weight distribution test is a statistical test to observe the deviation of the number of 1's in a pseudorandom bit sequence from the binomial distribution. As mentioned there, a corresponding theoretical evaluation is possible for $\mathbb{F}_2$-linear PRNG, called the weight discrepancy test. The test computes the exact distribution of the number of 1's for the PRNG. Differently from other theoretical evaluations, the probabilistic meaning of the weight discrepancy test is clear.

6 Warnings on the Usage of PRNGs

6.1 Problem of the Initialization

Before generating a sequence, we need to set the initial state of the PRNG. This is the initialization. Often the users want to give a seed of (say) 32-bit integer. Then, for a large state generator, it is necessary to expand the integer to fill the state space, by using some function. Usually, a small PRNG is used for this purpose.

In a large scale simulation, a seed of 32-bit integer is often too small. Let N be the number of possible seeds (namely, $N = 2^{32}$ in the above case). The *birthday paradox* asserts that if the seeds are chosen uniformly at random $\sqrt{N}$ times, then the probability that two of them coincide is close to $1/2$, for large N. In the above case, if one initializes 2^{16} times, then the probability that two of them coincide is close to $1/2$. By addressing this problem, the 2002 version of the initialization in `mt19937ar.c` receives an array of arbitrary length as an initial seed.

Another warning is that many modern PRNGs have a defect in the initialization scheme. If the initial seeds are systematically chosen (for example, if the seeds are chosen to be $0, 100, 200, \dots$), then the output sequences are often strongly correlated. The first author was informed of this by Isaku Wada. Later, Wada, Kuramoto, Ashihara, and the first author investigated the 58 different PRNGs in the GNU Scientific Library, and found that 45 of them have such defects [Matsumoto et al. 2006].

Thus, initialization of PRNG with huge state space is expensive. Usually it is better to avoid non-necessary initialization, since an initializing routine can generate only a small fraction of the state space, and they may have an undesirable structure. As an example, if the initial state of MT19937 has too many 0's, then the tendency continues for long (20000 generations or so). The initializing routine of `mt19937ar.c` was chosen to avoid such initial values.

The WELL generator mentioned in §5.2.2 is robust to such initial states.

6.2 PRNGs for Parallel Computation

PRNG for many parallel processors has different difficulties. A naive idea is to assign a different type of PRNGs to each processor. One problem with this idea is the difficulty in preparing many different types of PRNG. Another problem is the portability (i.e., the independence of the architecture): we sometimes want to reproduce a simulation done in a super parallel machine, using a moderately parallelized machine. Thus, we want the parallelization of PRNGs independent of the number of processors. To this end, each subject consuming the random numbers (i.e., each particle in a nuclear simulation) is assigned a unique ID, and a PRNG characterized by the ID is assigned to each.

Classically, only one PRNG is chosen, and the initial seed is generated from the ID. But this is dangerous, since the output sequences may be correlated as mentioned in §6.1, or may be overlapped.

A reasonable compromise is to use one and the same type of recursion, and to use a distinct parameter in the recursion for each PRNG. This method is called parameterization (e.g., SPRNG [Mascagni]).

[Matsumoto and Nishimura 2000] developed a C-program which receives ID and period, and computes a small type of MT with the ID embedded in its recursion formula (Dynamic Creator, freely delivered from [Matsumoto]).

Acknowledgments

This study is partially supported by the JSPS Core-to-Core Program #18005, by the JSPS Grant-In-Aid #16204002 and #18654021.

References

[Blum et al. 1986] L. Blum, M. Blum, and M. Shub. A simple unpredictable pseudo-random number generator. SIAM J. Comput. 15 (1986), 364–383.

[Couture et al. 1993] R. Couture, P. L'Ecuyer, and S. Tezuka, On the distribution of k-dimensional vectors for simple and combined Tausworthe sequences, Math. Comp. 60 (1993), 749–761.

[Ferrenberg et al. 1992] Ferrenberg, A. M., Landau, D. P., and Wong, Y. J. (1992) Monte Carlo simulations: hidden errors from 'good' random number generators. Phys. Rev. Lett. **69** 3382–3384.

[Fredricsson 1975] Fredricsson, S. A. (1975) Pseudo-randomness properties of binary shift register sequences. IEEE Trans. Inform. Theory **IT-21**, 115–120.

[Knuth 1997] Knuth, D. E. The Art of Computer Programming. Vol. 2. Seminumerical Algorithms 3rd Ed. Addison-Wesley, 1997.

[L'Ecuyer and Simard] L'Ecuyer, P., and Simard, R. TestU01 Empirical Testing of Random Number Generators, `http://www.iro.umontreal.ca/~simardr/indexe.html`

[Lewis and Payne 1973] Lewis, T. G., and Payne, W. H. Generalized feedback shift register pseudorandom number algorithms. *J. ACM* 20, 3 (1973), 456–468.

[Lindholm 1968] Lindholm, J. H. (1968) An analysis of the pseudo-randomness properties of subsequences of long m-sequences. IEEE Trans. Inform. Theory **IT-14**, 569–576.

[Lüscher 1994] Lüscher, M. A portable high-quality random number generator for lattice field theory simulations, Computer Physics Communications, 79 (1994) 100–110.

[Marsaglia] Marsaglia, G. `http://stat.fsu.edu/pub/diehard/`.

[Mascagni] Mascagni, M. The Scalable Parallel Random Number Generators Library (SPRNG), `http://sprng.cs.fsu.edu/`.

[Matsumoto] Matsumoto, M. Homepage of Mersenne Twister `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`.

[Matsumoto and Kurita 1992] Matsumoto, M. and Kurita, Y. "Twisted GFSR Generators," ACM Transactions on Modeling and Computer Simulation **2** (1992), 179–194.

[Matsumoto and Kurita 1994] Matsumoto, M. and Kurita, Y. "Twisted GFSR Generators II," ACM Transactions on Modeling and Computer Simulation **4** (1994), 254–266.

[Matsumoto and Nishimura 1998] Matsumoto, M. and Nishimura, T. "Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator" ACM Trans. on Modeling and Computer Simulation **8** (1998), 3–30.

[Matsumoto and Nishimura 2000] Matsumoto, M. and Nishimura, T. "Dynamic Creation of Pseudorandom number generator," 56–69 in: Monte Carlo and Quasi-Monte Carlo Methods 1998, Ed. H. Niederreiter and J. Spanier, Springer 2000. `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dc.html`

[Matsumoto and Nishimura 2002] Matsumoto, M. and Nishimura, T. “A Nonempirical Test on the Weight of Pseudorandom Number Generators” 381–395 in: Monte Carlo and Quasi-Monte Carlo methods 2000, Springer-Verlag 2002.

[Matsumoto and Nishimura 2003] Matsumoto, M. and Nishimura, T. “Sum-discrepancy test on pseudorandom number generators” Mathematics and Computers in Simulation, Vol. 62 (2003), pp 431-442.

[Matsumoto et al. 2006] Matsumoto, M., Wada, I., Kuramoto, A. and Ashihara, H. “Common Defects in Initialization of Pseudorandom Number Generators” submitted.

[Niederreiter 1992] Niederreiter, H. Random Number Generation and Quasi-Monte Carlo Methods. SIAM, 1992.

[Panneton et al. 2006] Panneton, F., L’Ecuyer, P. and Matsumoto, M. “Improved Long-Period Generators Based on Linear Reccurences Modulo 2” To Appear in ACM Transactions on Mathematical Software.