To appear in the *Annals of Operations Research (1994)*
Version: Fall 1993

UNIFORM RANDOM NUMBER GENERATION

Pierre L'ECUYER

Département d'IRO, Université de Montréal, C.P. 6128, Succ.A, Montréal, H3C 3J7, Canada

Abstract

In typical stochastic simulations, randomness is produced by generating a sequence of independent uniform variates (usually real-valued between 0 and 1, or integer-valued in some interval) and transforming them in an appropriate way. In this paper, we examine practical ways of generating (deterministic approximations to) such uniform variates on a computer. We compare them in terms of ease of implementation, efficiency, theoretical support, and statistical robustness. We look in particular at several classes of generators, such as linear congruential, multiple recursive, digital multistep, Tausworthe, lagged-Fibonacci, generalized feedback shift register, matrix, linear congruential over fields of formal series, and combined generators, and show how all of them can be analyzed in terms of their lattice structure. We also mention other classes of generators, like non-linear generators, discuss other kinds of theoretical and empirical statistical tests, and give a bibliographic survey of the most recent papers on the subject.

Keywords: Simulation, random number generation, pseudorandom, uniform random numbers, linear congruential, lattice structure, discrepancy, nonlinear generators, combined generators

1. Introduction

Simulating stochastic models requires a source of randomness. Modelers and programmers normally assume that random variables from different probability distributions, like normal, exponential, Bernouilli, Poisson, and so on, are available. The methods that have been designed to generate those random variables (see, e.g., Bratley, Fox, and Schrage [10], Devroye [19], or Law and Kelton [47]) assume in turn the availability of a source of i.i.d. $U(0,1)$, i.e., continuous random variables distributed uniformly over the real interval $(0,1)$, or sometimes discrete random variables distributed uniformly over some finite set of integers (like, e.g., random bits). In this paper, we discuss the methods which are most widely used, or most promising, for generating sequences of values which try to *imitate* such uniform random variables for simulation purposes. Those sequences are called *pseudorandom* and the programs which produce them are called *pseudorandom number generators.* In most of this paper, we will just use the term "random" instead of "pseudorandom", a slight abuse of language which is common usage in simulation contexts.

1.1. TRULY RANDOM VS PSEUDORANDOM SEQUENCES

In lotteries with prize money, the winning number is usually (hopefully) a *truly* random number, each digit being determined, for example, by physically drawing a (numbered) ball from some kind of container. This is not very practical for computer simulation, especially when millions of random numbers are required, as is often the case. Using truly random numbers for simulation and other Monte Carlo methods has been tried for a while, a few decades ago, but practically abandoned for various reasons [10, 49]. On modern computers, pseudorandom numbers are generated by completely deterministic algorithms. We want these numbers to look, from the outside, as if they where truly random. We would be pretty much happy if, for example, nobody who observes only the output sequence (and does not know the structure of the generator) can distinguish it from a truly random sequence in "feasible" time (say, a few years of cpu time on a large computer) better than by flipping a fair coin. In practice, however, most of the generators we use do not have such strong properties. This is still okay for many practical situations, but not all. There are reasonable applications for which many of the generators currently available on computers are useless (or dangerous). For example, for applications dealing with the geometrical behavior of random vectors in high dimensions, many generators must be avoided because of the bad geometrical structure of the vectors of successive points that they produce [50]. For cryptology, most available generators are dangerous because there are efficient ways of predicting the next value, given the sequence of values already produced by the generator [9, 42, 45].

1.2. BAD AND DANGEROUS GENERATORS

There is a well developed body of theory concerning the construction and analysis of (pseudo)random number generators. Good introductory references and survey papers include [6, 10, 16, 41, 44, 47, 49, 50, 75, 76, 77, 82]. Unfortunately, practice does not always keep up with theory. Many of the "default" generators currently offered in popular computer softwares, or suggested in some simulation textbooks, are old ones, and are not competitive with those based on the more recent theory. Much worse, many bad generators are still proposed every year in (supposedly serious) journal articles. One of my favorite exercises for students when I teach a simulation course is to have them test a bad generator recently proposed in a journal or available on a popular computer. For more on bad generators, see [41, 50, 79, 83]. As Ripley [84] said: "Random number generation seems to be one of the most misunderstood subjects in computer science". On the surface, it looks easy and attractive. This is probably why so many new generators are proposed by people from so many different fields (mathematics, computer science, physics, electrical engineering, management science, etc.). But building good generators is not so easy and requires a good understanding of the theory. As Knuth [44, page 5] said: "Random numbers should not be generated with a method chosen at random".

1.3. A DEFINITION OF A GENERATOR

Today's practical random number generators are computer programs which produce a deterministic, periodic sequence of numbers. The following definition is a slight variation from L'Ecuyer [49].

DEFINITION 1.

A *generator* is a structure $\mathcal{G} = (S, s_0, T, U, G)$, where S is a finite set of *states*, $s_0 \in S$ is the *initial state*, $T : S \to S$ is the *transition function*, U is a finite set of *output* symbols, and $G : S \to U$ is the *output function*.

A generator operates as follows: Start from the initial state s_0 (called the *seed*) and let $u_0 := G(s_0)$. Then, for $i := 1, 2, \ldots$, let $s_i := T(s_{i-1})$ and $u_i := G(s_i)$. We assume that efficient procedures are available to compute T and G. The sequence $\{u_i\}$ is the output of the generator and the u_i's are called the *observations*. For pseudorandom number generators, one would expect the observations to behave from the outside as if they were the values of i.i.d. random variables, uniformly distributed over U. The set U is often a set of integers of the form $\{0, \ldots, m-1\}$, or a finite set of values between 0 and 1 to approximate the $U(0, 1)$ distribution.

1.4. PERIOD AND TRANSIENT

Since S is finite, the sequence of states is ultimately periodic. The *period* is the smallest positive integer ρ such that for some integer $\tau \geq 0$ and for all $n \geq \tau$, $s_{\rho+n} = s_n$. The smallest τ with this property is called the *transient*. When $\tau = 0$, the sequence is said to be *purely periodic*.

1.5. QUASI-RANDOM SEQUENCES

The aim of a generator is not always to imitate true randomness as closely as possible. For example, in Monte Carlo numerical integration, one can take a sample of points over the domain of integration, and use the average of the function values at those points, multiplied by the volume of the integration domain, as an estimator of the integral. This can be done by using a pseudorandom sequence. But in terms of bounds on the integration error, one can often do better if the sample points are spread *more evenly* over the integration domain than a typical sample from the uniform distribution. The so-called *quasi-Monte Carlo* methods construct generators (in the sense of Definition 1) which produce deterministic sequences whose purpose is not to look random, but to give the best possible deterministic bounds on the integration error. Such sequences are called *quasi-random*. Bounds on the integration error can be obtained in terms of the *discrepancy* of the sequence (we will briefly explain that concept later on) and of some measure of variability of the function. Then, one looks for quasi-random sequences with the lowest possible discrepancy (or, in practice, with the lowest upper bound on their discrepancy, since the discrepancy can rarely be computed exactly). Niederreiter [77] is an excellent (high-level) introduction to quasi-Monte Carlo methods. In this paper, we will not enter further into that subject. Our interest will be in pseudorandom sequences.

1.6. A LITTLE BIT OF TRUE RANDOMNESS

In our definition of generator (Definition 1), the initial state s_0 was assumed to be given (deterministic). To introduce some real randomness, one can choose this initial state randomly, say by drawing balls from a box. In other words, we can generalize our definition by saying that the initial state s_0 is generated randomly according to some probability distribution μ on S. Generating a truly random seed is much less work and is more reasonable than generating a long sequence of truly random numbers. A generator with a random seed can be viewed as an *extensor* of randomness, whose purpose is to save "coin tosses". It stretches a short truly random seed into a long sequence of values that is supposed to appear and behave like a true random sequence.

1.7. OVERVIEW OF THE PAPER

In the next section, we discuss what we think are the properties that a good general purpose generator must possess: good statistical properties; long period; speed; low memory; portability; reproducibility; and splitting facilities. In §3, we define different classes of generators based on linear recurrences over some finite space (often a finite field). This will be our framework for most of what will follow afterwards. In §4, we give full-period conditions for recurrences over finite fields and discuss their verification in practice. We also look at the period lengths of other classes of generators. In §5, we examine the lattice structure of different classes of generators defined in §3. Some have a lattice structure in real space $\mathbb{R}^t$, while others have a lattice structure in a vector space of formal series. In both cases, the lattice structure characterizes how well the (overlapping) vectors of t successive values produced by the generator, over its entire period, are evenly distributed over the t-dimensional unit hypercube. We briefly discuss the notion of discrepancy in §6. Practical implementation considerations, especially for linear congruential, multiple recursive, and Tausworthe generators, are discussed in §7. In §8, we address the question of parallel generators and explain how to implement jumping-ahead (and splitting) facilities. The most popular approach for trying to improve the quality of generators is by combination of many different generators. This is the subject of §9. We discuss in more details two classes of combination approaches which have been recently analyzed successfully. §10 is about nonlinear generators, which do not have the same kind of lattice structure as the linear ones and have better discrepancy properties, but which are also slower. The question of empirical statistical testing is treated in §11.

2. What is a Good generator ?

We summarize in this section the major requirements for a good random number generator, for general purpose simulation. These requirements are also discussed in [10, 41, 79, 84], and we do not always share the views of all these authors.

2.1. STATISTICAL UNIFORMITY AND UNPREDICTABILITY

As we said, the sequence of observations from a generator should behave as if it was the realization of a sequence of independent random variables, uniformly distributed over the set U. But what does the word "behave" mean exactly here, and how can we verify whether the sequence behaves satisfactorily ? Various definitions of "random" sequence are given in Knuth [44]. These definitions apply to infinite, non-periodic, sequences, whereas the practical generators produce periodic sequences. From a pragmatic point of view, we can say that the generator should pass statistical tests for uniformity and independence. But after more careful thinking, we find out that this is a meaningless requirement. Indeed, since the sequence is deterministic and periodic, we know in advance that it is not truly uniform. In other words, we know that it is always possible to build a statistical test powerful enough, if enough time is allowed, so that the generator will fail it miserably. This looks like an hopeless situation.

One way out of this apparent deadend is to consider the time it takes to apply the tests in practice. We know that there is a test that can catch our generator. But if running that test requires billions of years of CPU time on the most powerful computers, then perhaps we don't care about that test. In other words, we might feel happy if the generator passes all (or almost all) the tests which can be run in "reasonable" time. This can be made more precise by using the ideas of computational complexity. The following definition is from L'Ecuyer and Proulx [54].

Consider a family $\{G_k,\ k \geq 1\}$ of generators, where k represents the size (e.g., the number of bits to represent the state). The family is called *PT-perfect* (polynomial-time perfect) if G_k "runs" in polynomial-time (in k) and if any polynomial-time (in k) statistical test which tries to distinguish the output of the generator from a truly random sequence, and to guess which is which, will not make the right guess with a probability larger than $1/2+\epsilon_k$, where $\epsilon_k,\ k \geq 1$ converges to zero exponentially fast. An equivalent definition is that no polynomial-time algorithm should be able to predict successfully u_{i+1} from $(u_0, \ldots, u_i)$ with a probability larger than $1/|U| + \epsilon_k$, i.e., significantly better than by picking a value uniformly from U. So, by taking k large enough, one has a safe generator which would pass all the statistical tests that can be run in reasonable time. For further details on these notions, see [8, 49, 54] and the references given there. The idea of PT-perfect generators was introduced by cryptologists, for which "unpredictability" is a crucial property. All of this looks nice, but the bad news are that no generator (family) has been proven PT-perfect to date. In fact, nobody even knows for sure whether there really exists any PT-perfect generator. Some generators *conjectured* to be PT-perfect have been proposed. However, they are still too slow for practical simulation use.

The generators mostly used in simulation (linear congruential, multiple recursive, GFSR, ...) are known *not* to be PT-perfect. "Efficient" algorithms have been designed to infer their sequence by looking at the first few numbers [9, 45]. But in practice, they remain quite useful for simulation, mainly because of their speed. When their parameters are well chosen and only a small fraction of their period is used, they show good statistical behavior with respect to most reasonable empirical tests. Binary (or m-ary) expansions of algebraic numbers (roots of polynomials with integral coefficients) or of some transcendental numbers (including π) do not define PT-perfect generators either. Kannan et al. [42] give efficient algorithms to compute further digits given a long enough initial segment of the expansion.

So, seeking PT-perfect generators for simulation might be too demanding and we are back to a weaker definition of "reasonable statistical test". Current practice sets up standard batteries of tests and apply them to the generators [20, 48, 50, 60]. Ideally, the tests should be selected in relation with the target application. But this is not always (easily) feasible, especially for "general purpose" generators which are to be provided in software packages or libraries. The question of statistical testing is further discussed in §11.

2.2. THEORETICAL SUPPORT

Empirical testing is fine, but there are often better ways of understanding the behavior of a generator, by theoretical analysis. Properties like the period length, lattice structure (or lack thereof), discrepancy, equidistribution, etc., usually give better insight on how the generator behaves. Generators lacking strong and convincing theoretical support must be avoided. The right approach for selecting a generator is to first screen out generators on the basis of their theoretical properties, and then submit the retained ones to appropriate empirical tests. In the next few sections, we will look at some of those theoretical properties.

In most cases, however, the available theoretical results are valid only for the entire period. For example, we might know that in a given dimension t, the t-tuples of successive output values, over the entire period, are very evenly distributed in the t-dimensional unit hypercube $[0,1]^t$. But in practice, we should use only a tiny fraction of the generator's period. Good equidistribution over the whole period might improve our confidence in good statistical behavior over the fraction of the period that we use, but provides no proof of such good behavior. In fact, when we use a generator whose points are very evenly distributed over the whole period, we implicitly *hope* and assume that over the small fraction of the period that we use, the points look like a *random* (and not perfectly evenly distributed) set of points. Indeed, points that are too evenly distributed fail to imitate randomness as well as points whose distribution is too far from even. Intuitively, we may view the set of points P over the entire period as a finite (but large) *sample space*, and the (much smaller) set of points P_0 that we use as a "random sample" (without replacement) from this set. If the points of P are not well distributed in the hypercube $[0,1]^t$, then it is likely that the points of P_0 will not look random. But if the points of P are very evenly distributed in $[0,1]^t$ and P_0 contains only a "negligible" fraction of those points, then the points of P_0 are likely to look random, as long as they behave somewhat like a random sample from P. As an analogy, imagine you want to put 10^{10} balls in a box, each bearing a number from 1 to 10^5, then draw 1000 of them *without* replacement, to simulate a sample of 1000 i.i.d. uniform variates from the set $\{1,\ldots,10^5\}$. Then, the best you can do is to put 10^5 balls of each number in the box. Of course, if you draw a sample of size close to 10^{10} instead of 1000, the set of numbers you will get will look *too* uniform (or "super-uniform"), but if you draw just a few, you will get nice random looking numbers.

However, we must be aware that this intuitive analogy has a catch: the points P_0 that we generate over a tiny part of the period are *not* a random sample, because the generator is deterministic. It may happen that in some dimension t, the points are generated in such an order that small subsets of successive points do not look random at all. So, having the points very evenly distributed over the entire period is appealing, but not enough. Theoretical results do not always apply only to the entire period; sometimes we can characterize the behavior of shorter subsequences as well. But in general, albeit necessary, theoretical support is not sufficient and should be supplemented by other empirical tests.

2.3. PERIOD LENGTH

As computers become increasingly faster, people perform larger simulations, which require more and more random numbers. Generators whose period length was sufficient some years ago are now unacceptable. For example, the period length of a multiplicative linear congruential generator with modulus 2^{32}, which is $\rho = 2^{30}$, can be exhausted in a few minutes of CPU time of a small workstation. Acceptable generators should have *at least* a period length of 2^{60} or more, and a much larger value is probably safer.

For most linear-type generators, the discrepancy of the vectors of successive values over the entire period is much too small compared with the discrepancy of truly random sequences (see §6 and [77]). Therefore, at most a small fraction of the period should be used. This gives further motivation for very long period generators. Based on a "nearest pair" argument, Ripley [82, p. 26] suggests that for linear congruential generators, the period (and the modulus) should always be at least an order of magnitude larger than the square of the number of values we use. Further, in many simulation applications, the generator's sequence is "split" into a large number of (disjoint) substreams, which should behave themselves as virtual generators (see [10, 52] and §8). Then, the period must be orders of magnitude longer. Families of fast-speed low-memory generators, with period lengths well over 2^{200}, have been proposed and analyzed recently and will be discussed in this paper.

2.4. EFFICIENCY

Despite the dramatic increases in computing power, speed and memory usage are still major concerns regarding generators. The time and memory space used by the random number generator might be insignificant in some cases [41], but (i) this will usually not be the case if the generator is slow or requires a lot of memory and (ii) there are cases where the time and space used by even the most efficient generators cannot be neglected [21, 43, 84]. Memory frugality becomes especially important when many "virtual" generators (i.e., many substreams) are maintained in parallel on a single computer (see §8).

2.5. REPEATABILITY, PORTABILITY, JUMPING AHEAD, AND EASE OF IMPLEMENTATION

A generator must be easy to implement efficiently in a standard high-level language. The code must be *portable*, i.e., produce exactly the same sequence (at least up to machine accuracy) with all "standard" compilers and on all "reasonable" computers. There is no good reason for choosing a generator which can be implemented only in machine-dependent assembly language. We do not say that a generator should never be implemented in assembly language, but at least, a high-level portable implementation must be available. Being able to reproduce the same sequence of random numbers on a given computer (called *repeatability*) is important for program verification and for variance reduction purposes [10, 47, 84]. Reproducing the same sequence on different computers is also important, for example for program verification across computers [41]. Repeatability is a major advantage of pseudorandom sequences with respect to sequences generated by physical devices. Of course, for the latter, one could store an extremely long sequence on a large disk or tape, and reuse it as needed thereafter. But this is not as convenient as a good pseudorandom number generator, which can stand in a few lines of code.

In our mind, ease of implementation also means the ease of breaking up the sequence into long disjoint substreams and jump ahead quickly from one substream to the other (see §8). This means that given the state s_n, it should be possible to calculate quickly the state $s_{n+\nu}$ for any large ν (without generating all the intermediate states, of course). Most "classical" (linear) generators allow such "leapfrogging" (even though the appropriate software tools are rarely available in packages or libraries). But there are classes of nonlinear and combined generators for which efficient ways of jumping ahead are unknown. One should think twice before selecting the backbone generator of a simulation package from such a class.

3. Generators Based on Linear Recurrences over Finite Fields

3.1. GENERAL FRAMEWORK

Most of the random number generators used in practice can be expressed by linear recurrences in modular arithmetic, over a finite set S. Often, S is a finite field and the transition function has the form $T(s) = \alpha s$, where $\alpha, s \in S$. In the latter case, S has the form $S = \mathbb{F}_{m^k}$, where $m = p^e$, p is prime, and e, k are positive integers. For $k = 1$, one has

$S = \mathbb{F}_m$, the finite field with m elements. Recall that $\mathbb{F}_m$ exists if and only if m is a power of a prime. When m is prime, one can identify $\mathbb{F}_m$ with the set $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$ on which arithmetic operations are performed modulo m. For $k \geq 1$, $\mathbb{F}_{m^k}$ can be constructed as a factor ring $\mathbb{F}_m[x]/(P)$, which can be identified with the space of polynomials modulo P and with coefficients in $\mathbb{F}_m$, where P is an irreducible polynomial of degree k with coefficients in $\mathbb{F}_m$. The state space S can also be viewed as the k-dimensional vector space $\mathbb{F}_m^k$ (space of k-dimensional vectors with elements in $\mathbb{F}_m$). A good reference on finite fields and related topics is Lidl and Niederreiter [58].

In what follows, unless otherwise indicated, we will assume that m is prime and that $\alpha \in S = \mathbb{F}_{m^k}$. Then, the state s_n of the generator evolves in $\mathbb{F}_{m^k}$ as

$$s_n = \alpha s_{n-1}. \tag{1}$$

Let

$$P(z) = z^k - a_1 z^{k-1} - \cdots - a_k \ \in \mathbb{F}_m[z]$$

be the minimal polynomial of α over $\mathbb{F}_m$. Then, in $\mathbb{F}_{m^k}$, one has $P(\alpha) = 0$, i.e.,

$$\alpha^n = a_1 \alpha^{n-1} + \cdots + a_k \alpha^{n-k}. \tag{2}$$

The value of k is called the *order* of the recurrence. If $P(z)$ is a primitive polynomial over $\mathbb{F}_m$ and α is a generator of the cyclic group $\mathbb{F}_{m^k}^* = \mathbb{F}_{m^k} \setminus \{0\}$, then the generator has *full period* $\rho = m^k - 1$, which means that if $s_0 \neq 0$, any subsequence of ρ consecutive values of s_n will visit each element of $\mathbb{F}_{m^k}^*$ once and only once (of course, $s = 0$ should not be visited, because it is an absorbing state). Further, $\alpha^\rho = \alpha^{m^k-1} = 1$ and $\alpha^{\rho-1} = \alpha^{-1}$ in $\mathbb{F}_{m^k}$.

Suppose that the output function is defined as a composition of the form $G = G_1 \circ G_2$, where $G_1 : \mathbb{F}_{m^k} \to \mathbb{F}_m$ is a linear form over $\mathbb{F}_{m^k}$, and $G_2 : \mathbb{F}_m \to [0, 1]$. This is the usual form of the output function in practice. Then, if $x_n = G_1(s_n) \in \mathbb{F}_m$, one has

$$x_n = a_1 x_{n-1} + \cdots + a_k x_{n-k} \tag{3}$$

in $\mathbb{F}_m$. Typically, we will directly implement the recurrence (3) over $\mathbb{F}_m$ instead of the recurrence (1) over $\mathbb{F}_{m^k}$. The transformation G_2 is sometimes defined by $G_2(x) = x/m$, where $x \in F_m$ is identified with its representative in $\mathbb{Z}_m$. The sequence $\{x_n\}$ is called a *linear recurring sequence* with *characteristic polynomial* $P(z)$. In fact, we will use that definition even when $P(z)$ is not primitive and even when m is neither a prime nor a power of a prime (in the latter case, the recurrence is in $\mathbb{Z}_m$, which is not a field). As we will see later on, some classes of generators based on linear recurrences with non-primitive (and reducible) characteristic polynomials, or linear recurrences modulo an integer m which has distinct prime factors, have very attractive practical properties.

To design a generator, typically, one selects m, k, U, and the output function G, then one finds a characteristic polynomial $P(z)$ of a desired form for which (1) can be implemented efficiently, and finally one tests the structural and statistical properties of the output sequence. In the remainder of this paper, we will discuss different ways of performing these tasks. We now examine a series of examples.

3.2. THE MULTIPLICATIVE LINEAR CONGRUENTIAL GENERATOR

Let $k = 1$, m prime, and identify $\mathbb{F}_m$ with $\mathbb{Z}_m$. Let $a = a_1 = \alpha \in \mathbb{Z}_m^* = \mathbb{Z}_m \setminus \{0\}$. Then (3) becomes

$$x_n = a x_{n-1} \bmod m. \tag{4}$$

If $G : \mathbb{Z}_m \to [0, 1]$ is defined by $G(x) = x/m$, this gives the classical *multiplicative linear congruential generator* (MLCG), which has been deeply analyzed, scrutinized, and often criticized, over the past 30 years or so [10, 33, 34, 44, 49, 77, 81]. Despite well founded critics, this kind of generator is still largely used in practice [10, 47]. One can also use (4) with a non-prime modulus m. Then, $\mathbb{Z}_m$ is not a field, but we still call the generator a MLCG. For example, m can be a large power of two. In that case, the characteristic polynomial $P(z) = z - a$ cannot be primitive and the largest possible period is only $m/4$, reached when $a \bmod 8 = 5$ and x_0 is odd. If m is prime, the period is $m - 1$ if and only if the *multiplier* a is a primitive root modulo m and $x_0 \neq 0$. Specific moduli and multipliers are analyzed in [33, 34, 43, 44, 48, 79].

3.3. THE MULTIPLE RECURSIVE GENERATOR

Let $k \geq 1$ and m prime. Again, identify $\mathbb{F}_m$ with $\mathbb{Z}_m$. The recurrence (3) is now

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod m. \tag{5}$$

The generator's state at step n is the vector $s_n = (x_n, \ldots, x_{n+k-1}) \in \mathbb{Z}_m^k$, which could be transformed into a value $u_n \in [0, 1]$ by $u_n = G(s_n) = x_n/m$. This kind of higher-order linear congruential generator is known as a *multiple recursive generator* (MRG) [38, 49, 51, 77]. The special case where $k = 1$ gives the usual MLCG with prime modulus. For $k > 1$, for $P(z)$ to be primitive, it is necessary that a_k and at least another a_j be non-zero. So, the most favorable case in terms of implementation is when $P(z)$ is a trinomial, of the form $P(z) = z^k - a_r z^{k-r} - a_k$. The recurrence (5) then becomes

$$x_n = (a_r x_{n-r} + a_k x_{n-k}) \bmod m. \tag{6}$$

3.4. DIGITAL MULTISTEP SEQUENCES AND THE TAUSWORTHE GENERATOR

Consider again the recurrence (5), for prime m, but redefine $s_n = (x_{ns}, \ldots, x_{ns+k-1})$ and

$$u_n = G(s_n) = \sum_{j=1}^{L} x_{ns+j-1} m^{-j}, \tag{7}$$

where s and $L \leq k$ are positive integers. Here, computing s_n from s_{n-1} involves performing s steps of the recurrence (5). Using a digital expansion in base m as in (7) yields a better resolution for the output values (for $L > 1$) than when u_n is just x_n/m. The output sequence $\{u_n\}$ obtained from such a generator is called a *digital multistep sequence* [75, 77] (Niederreiter [75, 77] imposes the additional constraints $L = s \leq k$, so our definition is more general). If (5) has full period $\rho = m^k - 1$ and s is coprime ρ, then the digital multistep sequence (7) also has period $\rho = m^k - 1$. Note that the previous example is a special case, with $s = L = 1$. Another important special case is when $m = p = 2$: the output values u_n are then constructed by taking blocks of L successive bits from the binary sequence (5) with spacings of $s - L \geq 0$ bits between the blocks. This was introduced by Tausworthe [88] and is known as a *Tausworthe* generator [44, 77, 90, 91]. (Sometimes, the Tausworthe generator is defined slightly differently, by filling up the bits of u_n from the least significant to the most significant one, instead of from most to least significant as in (7). See [88, 90]. This corresponds to generating the sequence (5) in reverse order.)

3.5. LINEAR RECURRENCES OVER SPACES OF POLYNOMIALS OR FORMAL SERIES

Let $P(z) = z^k - a_1 z^{k-1} - \cdots - a_k$ be a primitive polynomial over $\mathbb{F}_m$, $S = \mathbb{F}_{m^k} = \mathbb{F}_m[z]/(P)$ (the field of polynomials modulo $P(z)$, with coefficients in $\mathbb{F}_m$), and $\alpha \in S$ be a non-zero polynomial of the form $g(z) = (z^s \bmod P(z))$, where s is a positive integer. Observe that since $P(z)$ is primitive, any non-zero polynomial $g(z) \in S$ can be expressed as $g(z) = z^s \bmod P(z)$ for some integer s in the range $\{1, \ldots, m^k - 1\}$. Therefore, there is no loss of generality in imposing that form to $g(z)$. The state s_n at step n is a non-zero polynomial of degree smaller than k, with coefficients in $\mathbb{F}_m$. The transition function is given by

$$s_n(z) = z^s s_{n-1}(z) \bmod P(z), \tag{8}$$

where the arithmetic on the polynomial coefficients is performed in $\mathbb{F}_m$. Again, $\mathbb{F}_m$ can be identified with $\mathbb{Z}_m$.

If we formally divide $s_n(z)$ by $P(z)$, we obtain a formal Laurent series expansion in z^{-1}, with coefficients $d_{n,j} \in \mathbb{F}_m$:

$$\tilde{s}_n(z) = s_n(z)/P(z) = \sum_{j=1}^{\infty} d_{n,j} z^{-j}. \tag{9}$$

Dividing the equation (8) by $P(z)$, we see that this generator is in fact a linear congruential generator defined over the space of formal Laurent series:

$$\tilde{s}_n(z) = z^s \tilde{s}_{n-1}(z) \bmod \mathbb{F}_m[z]. \tag{10}$$

The multiplication by z^s in (10) corresponds to shifting the coefficients of $\tilde{s}_n(z)$ to the left by s positions, and the "mod $\mathbb{F}_m[z]$" operation means dropping off the terms with non-negative exponents in the formal series, i.e., those who were in the first s positions. In other words, one has $d_{n,j} = d_{n-1,j+s}$. Define $x_{j-1} = d_{0,j}$ for each $j \geq 0$. Then, $d_{n,j} = d_{0,ns+j} = x_{ns+j-1}$.

From the definition of $\tilde{s}_0(z)$, one obtains (replacing $d_{0,j}$ by x_{j-1}):

$$\begin{aligned} s_0(z) &= P(z)\tilde{s}_0(z) \\ &= \left(z^k - \sum_{i=1}^{k} a_i z^{k-i}\right)\left(\sum_{j=1}^{\infty} x_{j-1} z^{-j}\right) \end{aligned}$$

$$= \sum_{h=1}^{k} \left(x_{h-1} - \sum_{\ell=1}^{h-1} a_\ell x_{h-\ell-1} \right) z^{k-h}$$
$$+ \sum_{h=k+1}^{\infty} (x_{h-1} - a_1 x_{h-2} - \cdots - a_k x_{h-k-1}) z^{k-h}.$$

Since $s_0(z)$ is a polynomial, the coefficient of z^{k-h} must be zero for each $h > k$, i.e.,

$$x_j = (a_1 x_{j-1} + \cdots + a_k x_{j-k}) \bmod m$$

for each $j \geq k$. This is exactly the same recurrence as (5). Therefore, (8), (10), and (5) with $s_n = (x_{sn}, \ldots, x_{sn+k-1})$ (as in §3.4) are just different ways of expressing the same generator.

The above development of $s_0(z)$ also allows one to recover its coefficients from the sequence $\{x_n\}$. One can similarly recover the coefficients of $s_n(z)$ for any n: the coefficient of z^j in $s_n(z)$ is

$$\begin{aligned} c_{n,j} &= (x_{ns-j+k-1} - a_1 x_{ns-j+k-2} - \cdots - a_{h-1} x_{ns-j+k-h}) \bmod m \\ &= (a_h x_{ns-j+k-h-1} + \cdots + a_k x_{ns-j-1}) \bmod m. \end{aligned}$$

Replacing the formal variable z by the integer m in the formal series

$$\tilde{s}_0(z) = \sum_{j=1}^{\infty} d_{0,j} z^{-j} = \sum_{j=1}^{\infty} x_{j-1} z^{-j},$$

we obtain a digital fractional expansion in base m, namely

$$\tilde{u}_0 = \sum_{j=1}^{\infty} x_{j-1} m^{-j} = .x_0 x_1 x_2 \cdots. \tag{11}$$

Similarly, replacing z by m in $\tilde{s}_n(z)$ gives

$$\tilde{u}_n = \sum_{j=1}^{\infty} x_{ns+j-1} m^{-j} = .x_{ns} x_{ns+1} x_{ns+2} \cdots. \tag{12}$$

It is easily seen that $\tilde{u}_n$ is obtained by shifting the digital expansion of $\tilde{u}_0$ by ns positions to the left, and dropping the non-fractional digits. To produce the output u_n in practice, the digital expansion of $\tilde{u}_n$ can be truncated to, say, L digits. This yields

$$u_n = \sum_{j=1}^{L} x_{ns+j-1} m^{-j},$$

which is the same as (7). So, we have just recovered the digital multistep sequence by following a different development. This alternative view was first suggested by Tezuka and turns out to be quite useful for analyzing some of the structural properties of the sequence when m is small (e.g., $m = 2$) [14, 89, 90, 91]. The Tausworthe, MRG, and MLCG generators are special cases of this. In [91], for $m = 2$, the generator defined by (10–11) is called an LS2 generator.

3.6. MLCG'S IN MATRIX FORM

Let $P(z) = z^k - a_1 z^{k-1} - \cdots - a_k$ be a primitive polynomial and A a $k \times k$ matrix whose elements are in $\mathbb{F}_m$ and with characteristic polynomial $P(z)$. Consider the recurrence

$$X_n = AX_{n-1}, \tag{13}$$

where each X_n is a k-dimensional column vector of elements of $\mathbb{F}_m$ and the arithmetic is in $\mathbb{F}_m$. Then, it can be shown (see [37, 38, 71, 77]) that $\{X_n\}$ follows the recurrence

$$X_n = a_1 X_{n-1} + \ldots + a_k X_{n-k}. \tag{14}$$

In other words, each component of the vector X_n evolves according to (3), which means that we just have k copies of the same linear recurring sequence evolving in parallel, with perhaps different lags (or shiftings) between themselves (i.e., different initial states). Using (14) directly instead of (13) multiplies by k the size of the required memory, but often leads to quicker implementations (the state is then redefined as $s_n = (X_n, \ldots, X_{n+k-1})$). We will call (14) the *parallel MRG* implementation of the matrix generator (13). One instance of this is the GFSR generator, to be discussed later on.

Let us write X_n as

$$X_n = \begin{pmatrix} x_{n,1} \\ \vdots \\ x_{n,k} \end{pmatrix}.$$

Define $y_n = x_{n,1}$ and for $j > 1$, let d_j be the lag (or shift) associated with the component j of X_n. That is, $x_{n,j} = x_{n+d_j,1} = y_{n+d_j}$ for $2 \le j \le k$ and all $n \ge 0$. Those lags actually depend on the matrix A. One special case is when A is the companion matrix of $P(z)$:

$$A_c = \begin{pmatrix} 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \\ a_k & a_{k-1} & \ldots & a_1 \end{pmatrix}.$$

Then, the recurrences associated with the successive components of X_n are shifted one unit apart: $d_j = j - 1$ for all j. To have them shifted exactly d units apart, i.e., $d_j = (j-1)d$, one can just use $A = A_c^d$. In general, if $P(z)$ is the characteristic polynomial of A, one can write $A = PA_cP^{-1}$ for some regular matrix P. Then, $X_n = A^n X_0 = PA_c^n P^{-1} X_0$ for all n.

If the output of the matrix generator is produced by a composition of the form $G = G_1 \circ G_2$, where $G_1 : \mathbb{F}_{m^k} \to \mathbb{F}_m$ is linear, then the matrix generator is no more general than the MRG, in the sense that if we define $x_n = G_1(X_n)$, then the sequence $\{x_n\}$ obeys again the linear recurrence (5). In other words, any linear combination of elements of X_n in $\mathbb{F}_m$ obeys (5). However, there are other ways of combining the elements of X_n which might lead to a different recurrence than (5). We now examine two of them: the matrix MLCG, which uses each component X_n to produce k uniform variates per step, and the digital matrix MLCG, which uses the digital method on each X_n to produce one uniform variate per step.

Suppose that at each step n, each component of X_n is used to produce a uniform variate: that is, $u_{nk+j-1} = x_{n,j}/m$. This kind of matrix generator has been studied by Grothe [37, 38] and Afflerbach and Grothe [3]. We call it the *matrix MLCG*. It may prove useful for implementing parallel generators on parallel processors. One question arise: is the sequence $\{u_n\}$ produced by that generator the same as that produced by an MRG ? To answer that question, suppose that $\gcd(k, \rho) = 1$, where $\rho = m^k - 1$ is the period of $\{y_n\}$. Then, k has an inverse $d = k^{-1}$ in $\mathbb{Z}_\rho$, i.e., $kd \bmod \rho = 1$, and d can be easily computed via $d = k^{\rho-1} \bmod \rho$. Assume further that the lags are regularly spaced, d units apart: $d_j = (j-1)d$. Define $x_j = y_{jd}$, for $j \geq 0$. Since $\gcd(d, \rho) = 1$, $\{x_j\}$ is also a linear recurring sequence of period ρ. Further, for each $n \geq 0$, $x_{nk+j-1} = y_{nkd+(j-1)d} = y_{n+(j-1)d} = x_{n,j}$. So, the sequence $\{x_j, j \geq k\}$ is the same as the sequence obtained by taking all the components of all the vectors X_n in successive order: $x_{1,1}, \ldots, x_{1,k}, x_{2,1}, \ldots, x_{2,k}, \ldots$. Each of those components can be used to produce a uniform variate, e.g., as $u_n = x_n/m$. Note that $\{x_n\}$ is not necessarily a shift of $\{y_n\}$; in general, the (primitive) characteristic polynomials of those two sequences are different. Further, the above reasoning holds only if the d_j's are equally spaced $d = k^{-1}$ units apart.

Now, suppose that the output at step n is produced by the digital expansion

$$u_n = \sum_{j=1}^{L} x_{n,j} m^{-j} = \sum_{j=1}^{L} y_{n+d_j} m^{-j}, \tag{15}$$

where $L \leq k$. We call this generator a *digital matrix MLCG*. Again, let $\{y_n\}$ be a sequence with characteristic polynomial $P(z)$ and assume that the successive shifts between the first L components of $\{X_n\}$ are all equal: $d_j = (j-1)d$ for some positive integer d such that $\gcd(d, \rho) = 1$ (here, d is no longer the inverse of k). Then, d has an inverse in $\mathbb{Z}_\rho$, given by $s = d^{\rho-1} \bmod \rho$. Again, if we define $x_j = y_{jd}$, $\{x_j\}$ is a linear recurring sequence of period ρ and $x_{ns-k+j} = y_{nsd+(j-1)d} = y_{n+(j-1)d} = x_{n,j}$ for each $n \geq 0$. Then, (15) can be rewritten as

$$u_n = \sum_{j=1}^{L} y_{n+(j-1)d} m^{-j} = \sum_{j=1}^{L} x_{ns+j-1} m^{-j}. \tag{16}$$

This is the digital multistep sequence (7). Reciprocally, given a digital multistep sequence $\{x_n\}$ with $\gcd(s, \rho) = 1$, let $d = s^{\rho-1} \bmod \rho$ and consider a digital matrix MLCG with initial state given by $x_{0,j} = x_{j-1}$ for each j. Then, the sequence (15) produced by that digital matrix MLCG is the same as (7). In other words, a digital matrix MLCG, can be used for implementing (7).

3.7. LAGGED-FIBONACCI AND GFSR GENERATORS

An important special case of the digital matrix MLCG is when $m = 2$ and the generator is implemented using (14): this gives the well know Generalized Feedback Shift Register (GFSR) generator [36, 57, 77]. In that case, in (14), each X_n is a vector of bits and is obtained by making a bitwise exclusive-or of the X_{n-j}'s for which $a_j \neq 0$. Since only the first L bits of each X_n are used, one should keep only those first L bits. In practice, L is usually the word-size of the machine (e.g., $L = 32$), so that each X_n occupies one word of memory. The state $s_n = (X_n, \ldots, X_{n+k-1})$ at step n is a $L \times k$ dimensional array of bits. Practical GFSR generators are often based on a primitive trinomial: $P(z) = z^k - z^{k-r} - 1$. Then, one obtains

$$X_n = X_{n-r} \oplus X_{n-k},$$

where $\oplus$ denotes the bitwise exclusive-or.

The fact that a GFSR generator is equivalent to a Tausworthe generator when the initial state of the GFSR is chosen appropriately (with equally spaced shifts) gives us a good method for choosing that initial state: select a good Tausworthe generator and initialize the GFSR in such a way that it is equivalent to the Tausworthe (fill up the array of bits column by column using the Tausworthe generator). This approach is from Fushimi [35]. Tootill, Robinson, and Eagle [96] first suggested the GFSR implementation of Tausworthe sequences.

Observe that a GFSR generator (with parallel MRG implementation) can also be viewed as a "bigger" MLCG as follows. The state $s_n = (X_n, \ldots, X_{n+k-1})$ at step n is viewed as a kL-dimensional vector of bits $\boldsymbol{X}_n$. One has $\boldsymbol{X}_n = \boldsymbol{A}\boldsymbol{X}_{n-1}$ where

$$\boldsymbol{A} = A \otimes I = \begin{pmatrix} 0 & I & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & I \\ a_k I & a_{k-1} I & \ldots & a_1 I \end{pmatrix}, \qquad \boldsymbol{X}_n = \begin{pmatrix} X_n \\ X_{n+1} \\ \vdots \\ X_{n+k-1} \end{pmatrix},$$

I is the $L \times L$ identity matrix and $\otimes$ denotes the tensor product of matrices. We shall call that a GFSR in *expanded matrix form*. Of course, the characteristic polynomial of the $kL \times kL$ matrix $\boldsymbol{A}$ is not primitive over $\mathbb{F}_2$ and the generator does not reach the maximal possible period for a generator of that size, which is $2^{kL} - 1$. Its period length is only $2^k - 1$. One could then argue that the GFSR is a waste of memory [60, 61]. Ideally, a generator using kL bits of memory should have a period near 2^{kL}. This leads to the following idea: try to modify slightly the matrix $\boldsymbol{A}$ in such a way that it gets a primitive characteristic polynomial over $\mathbb{F}_2$, without impairing too much the speed of the GFSR generator. This is the subject of the next subsection. What we just said concerning GFSR generators also holds more generally for digital matrix generators over $\mathbb{F}_m$ instead of over $\mathbb{F}_2$, for m prime.

Some authors [72, 77, 90] use the expression *GFSR sequence* for the recurrence (15), whatever its implementation and (prime) value of m. We prefer to reserve the term GFSR to denote the parallel MRG implementation, because this is how the so-called GFSRs are actually implemented in practice, and to emphasize the size of the state space. If we extend our definition of GFSR to $m \neq 2$, then any digital matrix MLCG can be implemented as a GFSR. The converse is not true, however. For example, if $d_j = 0$ for all j, then X_n has all its components equal, for any n. If the GFSR has period $m^k - 1$ for $k > 1$, that cannot be implemented in the form of (13).

A "generalization" of the GFSR is the so-called *lagged-Fibonacci* generator, for which $\oplus$ can be replaced by any arithmetic or logical operation. One example is the *additive* generator [44], given by

$$X_n = (X_{n-r} + X_{n-k}) \bmod m,$$

where $m = 2^L$. This is a special case of the MRG, but with a power-of-two modulus. Its maximal period length, for suitable choices of r and k, is $(2^k - 1)2^{L-1} \approx 2^{k+L-1}$, which is 2^{L-1} times larger than that of a GFSR with the same values of L and k, but falls way short of 2^{kL}. See [60, 61] for more details and specific examples with the operators $+$, $-$, and $\times$, in arithmetic modulo 2^L.

3.8. TWISTED GFSR AND LARGE MATRIX GENERATORS

Matsumoto and Kurita [66] have proposed replacing $a_k I$ in the matrix $\boldsymbol{A}$ of the GFSR in expanded matrix form by an $L \times L$ matrix B of the form

$$B = \begin{pmatrix} 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \\ b_L & b_{L-1} & \dots & b_1 \end{pmatrix},$$

whose characteristic polynomial $P_B(z) = z^L - b_1 z^{L-1} - \cdots - b_L$ is such that $P_B(P(z) - a_k)$ is primitive over $\mathbb{F}_2$. The resulting matrix $\tilde{\boldsymbol{A}}$ then has a primitive characteristic polynomial of degree kL over $\mathbb{F}_2$, and the period becomes $\rho = 2^{kL} - 1$. For the usual case where the GFSR is based on a primitive trinomial $P(z) = z^k - z^{k-r} - 1$, $P_B(P(z))$ is primitive over $\mathbb{F}_2$ if and only if (i) $P_B(z)$ is irreducible and (ii) $\tilde{P}(z) = z^k - z^{k-r} - \eta$ is primitive over $\mathbb{F}_{2^L}$, where η is a root of $P_B(z)$ over $\mathbb{F}_2$. Matsumoto and Kurita explain how to find polynomials satisfying those conditions and call the resulting generators *twisted GFSR*, or TGFSR. In [67], they develop an improved variant of TGFSR generators (in terms of statistical behavior), which amounts to replacing B by PBP^{-1} for some well chosen regular matrix P. These TGFSR variants are fast, practically as fast as GFSR generators, and have extremely long period. They can be viewed as efficient ways of implementing digital matrix generators of order kL. Incidentally, the latter can also be implemented as large GFSRs (based on a characteristic polynomial of order kL).

Of course, one can do more than replacing only $a_k I$ by a more general matrix B; one can replace, say, each $a_j I$ in $\boldsymbol{A}$ by some more general matrix B_j in such a way that the resulting matrix $\tilde{\boldsymbol{A}}$ has a primitive characteristic polynomial. However, there is a compromise to be made in terms of implementation speed. If the B_j's do not have a special structure that can be exploited, the generator will be too slow.

3.9. ADD-WITH-CARRY AND SUBTRACT-WITH-BORROW GENERATORS

Marsaglia and Zaman [62] propose two types of random number generators, called *add-with-carry* (AWC) and *subtract-with-borrow* (SWB), which are slight modifications of the lagged-Fibonacci generators with the $+$ and $-$ operations, respectively. The AWC generator is based on the recurrence

$$\begin{aligned} x_j &= (x_{j-r} + x_{j-k} + c_j) \bmod b, && (17)\\ c_{j+1} &= I(x_{j-r} + x_{j-k} + c_j \geq b), && (18) \end{aligned}$$

where b and $k > r$ are positive integers, c_j is called the *carry*, and I is the indicator function, whose value is 1 if its argument is true, and 0 otherwise. This generator is similar to an MRG, except for the carry. It is extremely fast: it requires no multiplication and the modulo operation can be performed by just subtracting b when $c_{j+1} = 1$. The SWB has two variants. One is based on the recurrence:

$$\begin{aligned} x_j &= (x_{j-r} - x_{j-k} - c_j) \bmod b, &(19)\\ c_{j+1} &= I(x_{j-r} - x_{j-k} - c_j < 0), &(20)\end{aligned}$$

where $k > r$ and c_j is called the *borrow*. The other one is obtained by exchanging r and k in (19–20).

To produce the output, one can use L successive values of x_j for each u_n as in the digital multistep method:

$$u_n = \sum_{j=0}^{L-1} x_{Ln+j} b^{j-L}. \tag{21}$$

Note that here, the digits of u_n are filled up from the least significant to the most significant one.

Tezuka and L'Ecuyer [92] have shown that AWC and SWB generators are essentially equivalent to MLCGs in the following sense. Let $m = b^k + b^r - 1$ for AWC, and $m = b^k - b^r \pm 1$ (depending on the variant) for SWB. Suppose that m is prime and let $b^{-1} = b^{m-2} \bmod m$ be the multiplicative inverse of b modulo m. Let $a = (b^{-1})^L \bmod m = b^{(m-2)L} \bmod m$ and consider the MLCG:

$$y_n = a y_{n-1} \bmod m; \qquad\qquad w_n = y_n/m. \tag{22}$$

Then, Tezuka and L'Ecuyer prove that (assuming that y_0 is chosen appropriately to insure synchronization)

$$u_n = b^{-L} \lfloor b^L w_n \rfloor \tag{23}$$

for all $n > k$. In other words, u_n is w_n truncated to its first L fractional digits in base b. In other words, the sequences $\{u_n\}$ and $\{w_n\}$ are the same, if they have corresponding initial seeds, up to a precision of b^{-L}. For example, if $b \le 2^{30}$ and $L = 2$, then the first 60 bits of u_n and w_n are the same and for practical purposes, we may safely assume that $u_n = w_n$. So, we are back into the MLCG bandwagon.

The maximal period for the AWC or SWB is $m - 1$, which can be attained if a is a primitive element modulo m. With b around 2^{31} and k around 20, for example, one could reach a period of approximately 2^{620}. Marsaglia and Zaman suggest specific values of the parameters b, r, and s, most of them yielding extremely large periods. Unfortunately, as discussed in §5.1, these generators always have a bad lattice structure and therefore must be discarded.

4. Period Length and Primitive Polynomials

4.1. PRIME MODULUS

For a prime modulus m, the linear recurring sequence (5) has full period $\rho = m^k - 1$ if and only if its characteristic polynomial $P(z)$ is a primitive polynomial over $\mathbb{F}_m$. So, a first step in building a generator based on such a sequence is to find an appropriate primitive polynomial. How can we do that ? The following are necessary and sufficient conditions for $P(z)$ to be primitive over $\mathbb{F}_m$ (see Knuth [44]). Let $r = (m^k - 1)/(m - 1)$.

(a) $((-1)^{k+1}a_k)^{(m-1)/q} \neq 1$ for each prime factor q of $m - 1$;

(b) $(x^r \bmod P(z)) = (-1)^{k+1}a_k$;

(c) $(x^{r/q} \bmod P(z))$ has degree > 0 for each prime factor q of r, $1 < q < r$. ∎

The most difficult task in verifying those conditions is usually the factoring of r, unless r is prime. For that reason, and since testing primality is much easier than factoring, it is a good idea to choose m and k such that r is prime. For that, k must be odd. L'Ecuyer, Blouin, and Couture [51] discuss that topic and give some useful factorizations and primitive polynomials, for m near 2^{31}, 2^{47}, and 2^{63}. Given m, k, and the factorizations of $m - 1$ and r, it is relatively easy to find primitive polynomials simply by random search. In fact, there are exactly

$$N(m, k) = (m^k - 1)(1 - 1/q_1) \cdots (1 - 1/q_h)/k$$

vectors $(a_1, \ldots, a_k) \in \mathbb{F}_m^k$ that satisfy the conditions, where $q_1, \ldots, q_h$ are the distinct prime factors of $m^k - 1$ (see [44]). In the case $k = 1$, a primitive polynomial $x - a_1$ means that a_1 is a primitive element modulo m, and whenever one such a_1 has been found, all others can be found easily, since they are exactly all the integers of the form $a_1^j \bmod m$ where j is relatively prime to $m - 1$. For $k > 1$, it is often convenient to first find a value of a_k which satisfies condition (a), then perform a random search for the remaining coefficients $(a_1, \ldots, a_{k-1})$. Lists of primitive polynomials over $\mathbb{F}_2$ can be found in [39, 46, 58] and the references given there.

4.2. COMPOSITE MODULUS

When m is not prime, the maximal possible period for the linear recurring sequence (5) typically falls way short of $m^k - 1$. For $m = p^e$, p prime and $e \geq 1$, the upper bound is $(p^k - 1)p^{e-1}$, except for $p = 2$ and $k = 1$, where it is 2^{e-2} [30, 44]. Eichenauer-Herrmann, Grothe, and Lehn [30] show how to construct generators whose period reach that upper bound. For $p = 2$ and $k = 1$, see [44]. The case $p = 2$ is interesting in terms of efficiency, because the modulo operation can be implemented by just "chopping-off" the higher-order bits. However, it is costly in terms of period length. For example, if $k = 5$ and $m = 2^{31} - 1$, the maximal period length is $(2^{31} - 1)^5 - 1 \approx 2^{155}$, while if m is increased to 2^{31}, the longest possible period length becomes $(2^5 - 1)2^{31-1} \approx 2^{35}$. That is, approximately 2^{120} times shorter.

This is not the only reason why using a prime m is to be recommended. Another important reason is that for small p, the low order bits do not look random at all. For $p = 2$ and $k = 1$, the i-th least significant bit of x_n has period equal to $\max(1, 2^{i-2})$ [10, 18]. If the period of such a generator is split into 2^d equal segments, then all segments are identical except for their d most significant bits [18, 21]. For $i = 2^{e-d-2} > 0$, all points (x_n, x_{n+i}) lie on at most $\max(2, 2^{d-1})$ parallel lines [18]. For $k > 1$ (still with $p = 2$), the maximal period for the d-th least significant bit is $(2^k - 1)2^{d-1}$.

4.3. NON-MULTIPLICATIVE LCG'S

The MLCG is usually presented in the slightly more general form:

$$x_n = (ax_{n-1} + c) \bmod m, \tag{24}$$

where c is a constant. Adding such a constant permits one to reach a period length of m. Conditions for that are given in Knuth [44]. If m is prime, this has no real interest. Indeed, this just increases the maximal period by one, and otherwise gives no further significant improvement [44]. However, if m is not prime, e.g., if it is a power of two, then this more general form really becomes attractive. But it also has many drawbacks. For example, if $m = 2^e$, the period of the i-th least significant bit of x_n is at most 2^i and the pairs (x_n, x_{n+i}), for $i = 2^{e-d}$, lie in at most $\max(2, 2^{d-1})$ parallel lines [18].

One can also add a constant term c to the MRG recurrence (5) or a constant matrix C to the matrix MLCG (13). However, it can be shown [49] that any k-th order recurrence with such a constant term is equivalent to some $(k+1)$-th order MRG with no constant term. Therefore, a general upper bound on the period length if $m = p^e$ is $(p^{k+1} - 1)p^{e-1}$. Again, for large e and k, this is much smaller than m^k. All of these reasons argue against the use of power of two moduli.

5. Lattice Structure

5.1. THE LATTICE STRUCTURE OF MRG'S IN $\mathbb{R}^t$

It is well known that the vectors of successive values produced by a MLCG or MRG, in any given dimension, have a lattice structure [38, 44, 49, 81]. More precisely, consider the MRG (5). For any integer $t > 0$, let

$$T_t = \{\boldsymbol{u}_n = (u_n, \ldots, u_{n+t-1}) \mid n \geq 0,\ s_0 = (x_0, \ldots, x_{k-1}) \in \mathbb{Z}_m^k\}, \tag{25}$$

the set of all possible overlapping t-tuples of successive values produced by (5) with $u_n = x_n/m$, from all possible initial seeds. The set T_t turns out to be the intersection of a lattice L_t with the t-dimensional unit hypercube $I^t = [0,1)^t$. Recall that a d-dimensional *lattice* in $\mathbb{R}^t$ (for $d \leq t$) is a set of the form

$$L = \left\{ V = \sum_{j=1}^{d} z_j V_j \mid \text{ each } z_j \in \mathbb{Z} \right\},$$

where $V_1, \ldots, V_d \in \mathbb{R}^t$ is a set of independent vectors called a basis. The lattice L_t is usually t-dimensional (otherwise, all the points of T_t are contained in one hyperplane). A basis for L_t, as well as for its dual lattice, can be constructed as explained in [44] for $k = 1$ and in [38, 53] for $k \geq 1$. For $t \leq k$, one obviously has $L_t = \mathbb{Z}^t/m$, because $(x_0, \ldots, x_{t-1})$ can take any value in $\mathbb{Z}_m^t$, and so $\boldsymbol{u}_n$ can take any value in $\mathbb{Z}_m^t/m = (\mathbb{Z}^t/m) \cap I^t$. For a full period MRG, the latter also holds even if we limit s_0 in the definition of T_t to a fixed (non-zero) vector of $\mathbb{Z}_m^k$, and then add the zero vector to T_t, because s_n runs through all k-dimensional vectors with components in $\mathbb{Z}_m$ (except the zero vector) over the generator's period. For $t > k$, the set T_t contains only a small fraction of $\mathbb{Z}_m^t/m$. That fraction is equal to m^{k-t}.

If the generator does not have full period and if one considers only the cycle that corresponds to a given initial seed s_0, then, in general, the points do not form a lattice, but are still a subset of the lattice defined above, and typically also *generate* that same lattice. There are cases, however, where the points over one cycle generate a *sublattice* or a *subgrid* of L_t. A *grid* in $\mathbb{R}^t$ is a shifted lattice, i.e., a set of the form $V_0 + L$ where $V_0 \in \mathbb{R}^t$ and L is a lattice. One should then analyze the appropriate sublattice or subgrid instead of analyzing L_t (see also [34, 44]). One important example is an MLCG for which m is a power of two, $a \bmod 8 = 5$, and x_0 is odd. In that case, as observed by Hoaglin and King [40] and Atkinson [7], the t-dimensional vectors of successive values form a subgrid of L_t containing one-fourth of the points. Another important case is when m is a product of J distinct primes $m = m_1 \cdots m_J$ (see [55] and §9). Then, in most cases of practical interest (according to my empirical experience), the generator has a few long subcycles of length $\rho = \mathrm{lcm}\,(m_1 - 1, \ldots, m_J - 1)$, plus some shorter subcycles, and the set of points over each of those long subcycles of length ρ typically generates the whole lattice L_t.

The fact that the points of T_t belong to a lattice means that they lie on a set of equidistant parallel hyperplanes. The shorter the distance d_t between those hyperplanes, the better, because this means thinner empty (without points) slices of space. Computer programs now exist for computing d_t in reasonably large dimensions, up to around 40 or more [53].

A slightly different way of measuring the "quality" of the lattice is by the Beyer quotient, defined as follows. Geometrically, the lattice L_t "partitions" the space $\mathbb{R}^t$ into a juxtaposition of identical t-dimensional parallelepipeds whose vertices are points of the lattice, and which contain no other lattice points except for their vertices. Those are called the *unit cells* of the lattice. The volume of each unit cell, called the *determinant* of the lattice, is one over the cardinality of T_t. All the edges connected to a given vertex of a unit cell form a set of linearly independent vectors which form a *lattice basis*. Such a basis is called *Minkowski-reduced* (MRLB) when the basis vectors are in some sense most orthogonal [2]. The *Beyer quotient* q_t is the ratio of the length of the shortest vector over the length of the longest vector in a MRLB. A ratio q_t near one means that the unit cells are more cubic-like and that the points are more evenly distributed, while a ratio near 0 means the opposite. In contrast to d_t, q_t is a normalized measure (always between 0 and 1). As a figure of merit to rank generators, one can take for example the worst-case measure

$$Q_T = \min_{t \leq T} q_t \tag{26}$$

for some (fixed) large T. However, q_t is much more costly to compute than d_t. An algorithm to compute a MRLB and the Beyer quotient is given in Afflerbach and Grothe [2]. See also [38, 53]. L'Ecuyer, Blouin, and Couture [51] suggest specific MRGs, after a search based on the criterion Q_{20}, for orders k up to 7 and prime moduli up to near 2^{63}. As observed by L'Ecuyer [49], comparing Beyer quotients makes sense only for generators having the same number of lattice points in the unit hypercube. A full period MRG has m^k such points, i.e., unit cells of volume m^{-k}, in all dimensions. For $t \leq k$, the lattice is a perfect square grid of size $1/m$, and the Beyer quotient is 1. Increasing m or k gives smaller unit cells. If a generator G_1 has smaller Beyer quotient than another generator G_2, then G_1 might still be better than G_2 if it has smaller unit cells. In such a situation, as a bottomline criterion, one can turn back to the distance d_t between hyperplanes. If G_1 has a smaller d_t than G_2 for all t, then we can say that G_1 *dominates* G_2 in terms of the spectral test, and claim that G_1 has a better lattice structure.

The matrix MLCG, which uses all the components of X_n at each iteration, has a similar lattice structure and can be analyzed in a similar way [3, 38]. When the MLCG is not multiplicative, the lattice is shifted by a constant vector, becoming a grid. The structure can be analyzed in the same way, since it does not really depend on the additive constant except for shifting. When T_t is replaced by the set of *non-overlapping* t-tuples, L_t does not form a lattice in general [1].

The AWC/SWB generators described in §3.9 are equivalent to linear congruential generators, and therefore have a lattice structure. In [14, 93], it is shown that this structure is always bad: in all dimensions $t > k$, one has $d_t \geq 1/\sqrt{3}$. It is also shown that combining an AWC/SWB generator with a LCG still yield an unfavorable lattice structure in large dimensions. That could explain the statistical anomalies observed empirically in [32, 50].

5.2. EQUIDISTRIBUTION WITH FINITE RESOLUTION AND LATTICE STRUCTURE IN THE SPACE OF FORMAL SERIES

Tausworthe and GFSR generators also have a lattice structure, which stems from the fact that they can be expressed as MLCGs over a space of formal series (see Equation (10)). To analyze the meaning of such a lattice structure, we will use the following definitions.

The (t,ℓ)-*equidissection in base* m of the t-dimensional unit hypercube I^t is a partition of I^t into $m^{t\ell}$ cubic cells of equal size. A finite set of points P in I^t is said to be (t,ℓ)-*equidistributed* in base m if each cell of the (t,ℓ)-equidissection contains the same number of points of P. When the value of the base m is clear from the context, we often omit the expression "in base m". In practice, the most interesting base is $m = 2$. Clearly, for P to be (t,ℓ)-equidistributed in base m, its cardinality must be a multiple of $m^{t\ell}$. In our applications, it will in fact be a power of $m^{t\ell}$. It is evident that (t,ℓ)-equidistribution implies (t',ℓ')-equidistribution for all $t' \leq t$ and $\ell' \leq \ell$. Furthermore, (t,ℓ)-equidistribution in base m is equivalent to $(t,1)$-equidistribution in base m^ℓ. Knuth [44, p.144] has a related definition: when the latter holds, he says that the m^ℓ-ary sequence is t-distributed.

Consider the set $\tilde{T}_t$ of all t-tuples of successive formal series obtained from (10):

$$\tilde{T}_t = \{\tilde{\boldsymbol{s}}_n = (\tilde{s}_n, \ldots, \tilde{s}_{n+t-1}) \mid n \geq 0, \tilde{s}_0 \in \tilde{S}\} \tag{27}$$

where $\tilde{S}$ is the set of formal series of the form

$$\tilde{S} = \{f(z)/P(z) \mid f(z) \text{ is a polynomial of degree } < k\}.$$

The mapping

$$\tilde{s}(z) = \sum_{j=1}^{\infty} c_j z^{-j} \longrightarrow \sum_{j=1}^{\infty} c_j m^{-j},$$

when applied componentwise, maps $\tilde{T}_t$ to a finite set of points $P_t \subset I^t$. These points are in fact all the t-dimensional vectors of successive values of (12), from all possible initial seeds $\tilde{s}_0 \in \tilde{S}$:

$$P_t = \left\{ \tilde{\boldsymbol{u}}_n = (\tilde{u}_n, \ldots, \tilde{u}_{n+t-1}) \mid n \geq 0, \tilde{s}_0 \in \tilde{S} \right\}.$$

We are interested in knowing how well those points are distributed in I^t. If P_t is (t,ℓ)-equidistributed, we say that the sequence (12) is (t,ℓ)-equidistributed. If $L \geq \ell$, the sequence (7) is also (t,ℓ)-equidistributed and we say that the generator is (t,ℓ)-equidistributed. In the best case (like for a full period generator), P_t has cardinality m^k, so (t,ℓ)-equidistribution is possible only for $\ell \leq \lfloor k/t \rfloor$. When the sequence is $(t, \lfloor k/t \rfloor)$-equidistributed for $t = 1, \ldots, k$, we say that it is *maximally equidistributed*. Some authors also call such a sequence *asymptotically random* [90, 96].

Full period digital multistep generators (7) are all $(1,k)$-equidistributed (for $L = k$), because each possible vector s_n (except zero) occurs once and only once over the full period. Tausworthe [88] also showed that they are $(\lfloor k/s \rfloor, s)$-equidistributed. Tootill, Robinson, and Eagle [96] found the following maximally equidistributed Tausworthe generator: $P(z) = z^{607} - z^{273} - 1$, $s = 512$, and $L = 23$. In a similar vein, all GFSRs based on primitive polynomials are $(k,1)$-equidistributed, because their first bit evolves according to a full period MRG of order k. But for more than one bit of resolution, the equidistribution properties of the GFSR depend on the lags d_j between the components of X_n, i.e., on the initial state $s_0 = (X_0, \ldots, X_{k-1})$. If the initial state is badly chosen, one might not even have $(1,2)$-equidistribution: for example, just take $d_1 = d_2 = 0$. Fushimi and Tezuka [36] gave a necessary and sufficient condition on the initial state for the GFSR generator to be (t,L)-equidistributed for $t = \lfloor k/L \rfloor$. Consider the tL bits $(x_{0,1}, \ldots, x_{0,L}, \ldots, x_{t-1,1}, \ldots, x_{t-1,L})$. The condition is that those bits must be independent in the sense that the $tL \times k$ matrix which expresses them as a linear transformation of $(y_0, \ldots, y_{k-1}) = (x_{0,1}, \ldots, x_{k-1,1})$ has (full) rank tL. Fushimi [35] gives a nice initialization procedure for GFSR generators to satisfy that condition, based on the use of an equivalent Tausworthe generator. Besides being slow and cumbersome, the GFSR initialization procedures previously available merely insured $(1,L)$-equidistribution. In fact, the condition for (t,L)-equidistribution can be used as well to verify (t,ℓ)-equidistribution for any $\ell \leq L$: just pretend that the word size is ℓ, i.e., replace L by ℓ. All of this also generalizes to $m \neq 2$.

The set $\tilde{T}_t$ generates the following lattice over the field of formal series:

$$\begin{aligned}
\tilde{L}_t &= (\mathbb{F}_m[z])\tilde{T}_t + (\mathbb{F}_m[z])^t \\
&= \left\{ g(z)\tilde{\boldsymbol{s}}(z) + (h_1(z), \ldots, h_t(z)) \mid \tilde{\boldsymbol{s}}(z) \in \tilde{T}_t \text{ and } g, h_i \in \mathbb{F}_m[z] \right\}.
\end{aligned}$$

where $\mathbb{F}_m[z]$ is the space of polynomials in z with coefficients in $\mathbb{F}_m$. Let $\mathbb{F}_m((z^{-1}))$ denote the space of formal Laurent series of the form $\tilde{s}(z) = \sum_{j=h}^{\infty} c_j z^{-j}$ and define a norm on the vector space $(\mathbb{F}_m((z^{-1})))^t$ as follows. For each $\tilde{\boldsymbol{s}} = (\tilde{s}_1, \ldots, \tilde{s}_t) \in (\mathbb{F}_m((z^{-1})))^t$, where $\tilde{s}_i(z) = \sum_{j=h_i}^{\infty} c_{i,j} z^{-j}$ with $c_{i,h_i} \neq 0$, define $\|\tilde{\boldsymbol{s}}\| = \max_{1\le i\le t} m^{-h_i}$. If all $c_{i,j}$'s are zero, define $\|\tilde{\boldsymbol{s}}\| = \|0\| = 0$. Using this norm to define distances, and assuming that $\tilde{L}_t$ has dimension t (which is usually the case), let $\mathcal{B} = \{\tilde{V}_1, \ldots, \tilde{V}_t\}$ be a set of vectors in $\tilde{L}_t$ such that $\tilde{V}_1$ is a shortest vector in $\tilde{L}_t$ and, for $j = 2, \ldots, t$, $\tilde{V}_j$ is a shortest vector in $\tilde{L}_t$ among those which are linearly independent of $\{\tilde{V}_1, \ldots, \tilde{V}_{j-1}\}$. Then, it can be shown [14] that $\mathcal{B}$ is a basis for $\tilde{L}_t$, and it is called a *reduced basis*. So, the lattice can be expressed as

$$\tilde{L}_t = \left\{ \sum_{j=1}^{t} g_j \tilde{V}_j \mid g_j \in \mathbb{F}_m[z] \right\}.$$

Reduced bases can be computed via Lenstra's algorithm [56]. The values $\|\tilde{V}_1\|, \ldots, \|\tilde{V}_t\|$ are called the *successive minima* of $\tilde{L}_t$. Define $\ell_j = -\log_m \|\tilde{V}_j\|$ and for each integer $\ell > 0$, let

$$d(\ell) = \sum_{j=1}^{t} (\ell_j - \ell)^+. \tag{28}$$

Now, assume that (8) has full period $\rho = m^k - 1$. For the case where $P(z)$ is irreducible, Couture, L'Ecuyer, and Tezuka [14] have shown that in the (t,ℓ)-equidissection in base m, $m^{k-d(\ell)}$ cells contain $m^{d(\ell)}$ points of P_t each, while $m^{t\ell} - m^{k-d(\ell)}$ cells contain no point of P_t. Therefore, P_t is (t,ℓ)-equidistributed if and only if $d(\ell) = k - t\ell$. (The proofs in [14] are given for $m = 2$, but their generalization to any prime m is straightforward.) If one considers only the main cycle of the generator, i.e., discards the zero formal series (as done in [14]), then the cell with one corner at the origin contains one point less. In fact one always has

$$\sum_{j=1}^{t} \ell_j = k.$$

Then, recalling that $\ell_1 \geq \cdots \geq \ell_t$, one can see that $d(\ell) = k - t\ell$ is equivalent to $\ell_t = \min_{1\le j\le t} \ell_j \geq \ell$. Therefore, ℓ_t gives the resolution of the generator, that is, the maximum value of ℓ for which it is (t,ℓ)-equidistributed. In other words, we want the minimum ℓ_j to be as large as possible. This is achieved if and only if $\ell_1 - \ell_t \geq 1$. So, ℓ_t and $\ell_1 - \ell_t$ act as respective analogues of the distance d_t between hyperplanes and the Beyer quotient q_t defined in §5.1.

For the case where $P(z)$ is reducible with J factors, Couture, L'Ecuyer, and Tezuka [14] have obtained general results giving a precise description of how the points of P_t are distributed into the cells of the (t,ℓ)-equidissection. For $J = 2$ and 3, they give explicit formulæ to quickly compute how many cells contain exactly n points, for each integer n, in terms of the successive minima of different lattices. They show how to construct bases for those lattices. From that, generators can be found which are "approximately" (t,ℓ_t)-equidistributed [14].

As we saw in §3.7, there is an equivalence between Tausworthe generators and GFSR generators with appropriate parameters and initial states. Therefore, the lattice structure of such GFSR generators can be analyzed in the same way as for Tausworthe generators. More general GFSR generators (with unevenly spaced shifts d_i) and twisted GFSR generators also have a lattice structure. See Tezuka [89, 90].

5.3. NETS AND NIEDERREITER'S FIGURE OF MERIT

A stronger notion than that of (t, ℓ)-equidistribution is the notion of net, introduced by Sobol' (see [77]). A *(q, k, t)-net in base m* is a set of m^k points in I^t such that each *elementary interval* E of I^t of the form

$$E = \prod_{i=1}^{t} [\alpha_i, \alpha_i + 1) m^{-\gamma_i}$$

where each α_i and γ_i are non-negative integers such that $\alpha_i m^{-\gamma_i} < 1$, and with volume m^{q-k} (i.e., $\sum_{i=1}^{t} \gamma_i = k - q$), contains exactly m^q points. In the case, of the (t, ℓ)-equidistribution, we were considering only the *cubic* elementary intervals E, i.e., we were imposing that all γ_i's be equal.

Consider a digital multistep sequence (7). Let $t > \lfloor k/s \rfloor$ and let α be a root of $P(z)$ in $\mathbb{F}_m^k$. Consider the set of vectors

$$C = \left\{ \alpha^{(i-1)s+j-1} \mid 1 \le i \le t, 1 \le j \le L \right\}.$$

Let ℓ^* be the largest integer such that $\ell^* = \sum_{i=1}^{t} \ell_i$, $0 \le \ell_i \le L$ for each i, and such that

$$C_1(\ell_1, \ldots, \ell_t) = \left\{ \alpha^{(i-1)s+j-1} \mid 1 \le i \le t, 1 \le j \le \ell_i \right\}$$

is linearly independent in $\mathbb{F}_m^k$. Niederreiter [77] defines the *figure of merit* $r^{(t)} = \min(L, \ell^*)$ and proves that for $t > \lfloor k/L \rfloor$, the m^k points of T_t form a (q, k, t)-net in base m with $q = k - r^{(t)}$. He also proves that the same holds for the digital matrix generator (15) if we replace C_1 by

$$C_2(\ell_1, \ldots, \ell_t) = \left\{ \alpha^{i+d_j-1} \mid 1 \le i \le t, 1 \le j \le \ell_i \right\}.$$

Note that $\ell^* \le k$ and $r^{(t)} \le \min(L, k)$ always hold. Assuming that we seek low discrepancy over the full period, the smaller q the better, i.e., we want $r^{(t)}$ to be as large as possible. In the best case, one has $r^{(t)} = k$, i.e., $q = 0$, and each elementary interval of the $(0, k, t)$-net contains exactly one point. According to Corollary 4.21 of [77], a $(0, k, t)$-net can only exist for $t \le m + 1$. Therefore, for $t > m + 1$, one must have $r^{(t)} \le k - 1$.

Unfortunately, finding generators with large $r^{(t)}$ for large k and t appears difficult for the moment, from the computational point of view. Steps in that direction have been made by André, Mullen, and Niederreiter [5]. For $m = 2$, they have computed a list of primitive polynomials of degree $k \le 32$ for which $r^{(2)} \ge k - 1$ and $r^{(t)}$ is large for all $t \le 5$. Tezuka and Fushimi [94] extended those results to a list of polynomials for which $r^{(2)} = k$ and $r^{(t)}$ is large for $t \le 6$. Their associated sequences can also be generated more quickly than those of [5], with the GFSR implementation.

6. Discrepancy and Other Theoretical Measures

The notion of *discrepancy* has been the subject of many papers and is well treated in the excellent book of Niederreiter [77], who is undoubtedly the "grand master" of the subject. Here, we just give it a quick look. For more details, see the many references given in Niederreiter [69, 75, 77].

Suppose we generate N t-dimensional points $\boldsymbol{u}_n = (u_n, \ldots, u_{n+t-1})$, $0 \le n \le N-1$, formed by (overlapping) vectors of t successive values produced by the generator. For any hyper-rectangular box aligned with the axes, of the form $R = \prod_{j=1}^{t} [\alpha_j,\ \beta_j)$, with $0 \le \alpha_j < \beta_j \le 1$, let $I(R)$ be the number of points $\boldsymbol{u}_n$ falling into R, and $V(R) = \prod_{j=1}^{t} (\beta_j - \alpha_j)$ be the volume of R. Let $\mathcal{R}$ be the set of all such regions R, and

$$D_N^{(t)} = \max_{R \in \mathcal{R}} |V(R) - I(R)/N|.$$

The latter is called the t-dimensional *(extreme) discrepancy* for the set of points $\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{N-1}$. If we impose $\alpha_j = 0$ for all j, i.e., we restrict $\mathcal{R}$ to those boxes which have one corner at the origin, we obtain a variant called the *star discrepancy*, denoted by $D_N^{*(t)}$.

Points whose distribution is far from uniform will have high discrepancy, while points which are too evenly distributed will tend to have a discrepancy that is too low. A well behaved generator should have its discrepancy in the same order (for large N) as that of a truly random sequence, which lies between $O(N^{-1/2})$ and $O(N^{-1/2}(\log\log N)^{1/2})$, according to the law of the iterated logarithm [76, 77]. This holds for both the star and extreme discrepancies. (Here, $O(f(n))$ denotes the *set* of functions g such that for some constant $c > 0$, $g(n) \le cf(n)$ for all n.) Niederreiter [77] shows that for a full period MLCG (with period $\rho = m-1$), for an *average* multiplier a (average over the set of multipliers which are primitive modulo m), the discrepancy $D_{m-1}^{(t)}$ over the full period is in $O(m^{-1}(\log m)^t \log\log(m+1))$. For large m, this is too small, meaning too much regularity. Niederreiter [76, 77] concludes that for that reason, MLCGs should be discarded altogether. A somewhat different interpretation (or conclusion) could be that in practice, one should never use more than a tiny fraction of the period of the MLCG. Because of the lattice structure, it is clear from the outset that over the full period, the points will be much too evenly distributed. This is even more so when the Beyer quotient q_t is close to 1. However, as explained in §2.2, super-uniformity over the entire period is reassuring and intuitively good when we use only a tiny fraction of the period. Bounds on the discrepancy also exist for part of the period [70] and the discrepancy is then better behaved. Of course, using only a small fraction of the period is not necessarily foolproof, but at least the argument of the wrong order of magnitude of the discrepancy no longer stands in that case.

Consider now the digital multistep method (7). The *resolution* here is m^{-L}, which means that all u_n's are rational with denominator m^L. From that, it is easily seen [77] that $D_N^{*(t)} \ge m^{-L}$ for all $t \ge 1$ and $N \ge 1$. Further, for $t \le k/s$ and $L = s = k$, Niederreiter [77] shows that $D_N^{*(t)} = 1 - (1 - m^{-L})^t$ for $N = \rho$ (the period). Therefore, a *necessary* condition for the discrepancy to be in the right order of magnitude is that the resolution m^{-L} must be small enough for the number of points N that we plan to generate. A too coarse discretization implies a too large discrepancy. If N points are to be used, one should take m^{-L} much smaller than $N^{-1/2}$. These recommendations apply in particular to MLCGs and MRGs (with $L = 1$) and Tausworthe (with $m = 2$). They also provide justification for using the digital method even when m is large. For $L = s = k$ and $t > k/s$, one has $D_N^{*(t)} \in O(r^{t-1} m^{-r})$, where $r = r^{(t)}$ is the figure of merit of the generator defined in §6.3 (see [77]). So, a larger figure of merit suggests a lower discrepancy. This also holds for digital matrix generators. Further, on the "average" (over primitive polynomials), for $N = m^k - 1$ (the period) and assuming again that $L = s = k$, one has $D_N^{*(t)} \in O(N^{-1}(\log N)^{t+1} \log\log N)$ for the digital multistep method and $D_N^{*(t)} \in O(N^{-1}(\log N)^t)$ for digital matrix generators. For large N, these discrepancies are too small. Therefore, the same recommendation as for MLCGs holds here: never use more than a tiny fraction of the period. One question arises here: since $D_N^{*(t)}$ is already too small on the average, and decreases with $r^{(t)}$, why should we seek a large figure of merit $r^{(t)}$? Again, as explained in §2.2, having the points $\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{\rho-1}$ very evenly distributed gives us (heuristically) greater confidence that the small fraction that we use will be random looking. We view the latter fraction of points somewhat like a random sample from the whole set. Discrepancy bounds for part of the period are given in [72, 73].

One major difficulty with discrepancy is that it can be computed exactly only for a few very special cases (e.g., for a LCG, for $t = 2$ [4]). Otherwise, only bounds on $D_N^{(t)}$, or orders of magnitude, are available [77]. Typically, these orders of magnitude are for N equal to the period length, or are averages over a whole class of generators. Estimating the discrepancy empirically, e.g., from a fine grid, does not seem possible for moderate t (say, $t \geq 4$) and reasonably large N. Another drawback is that discrepancy depends on the orientation of the axes, in contrast to the Beyer quotients and distance between hyperplanes. On the other hand, generators of different types (e.g., linear vs nonlinear) can be compared in terms of the order of magnitude of their discrepancies. This cannot be done with the lattice test. Finally, discrepancy is also interesting and useful because one can obtain error bounds for (Monte Carlo) numerical integration or random search procedures in terms of $D_N^{(t)}$. In that context, the smaller the discrepancy, the better (because the aim is to minimize the numerical error, not really to imitate i.i.d. $U(0,1)$ random variables).

There are other "statistical measures" which we did not discuss here and which can be computed exactly (or bounds for them can be computed) for specific classes of generators. That includes computing bounds on the serial correlation [44], computing the results of the "run" test applied over the whole sequence of a Tausworthe generator [95], computing the nearest pair of points over the whole period, or the minimal number of hyperplanes that cover all the points, etc.. See [44] for further details.

7. Implementation and Efficiency Considerations

7.1. LINEAR CONGRUENTIAL AND MRG GENERATORS

Implementing (5) in a portable way, in high level language, for a large prime modulus m, is tricky in general because of the possible overflow in the products. If m is representable as a standard integer on the target computer, there is a reasonably efficient and portable way of computing $ax \bmod m$ for $0 < x < m$ provided that

$$a(m \bmod a) < m. \tag{29}$$

See [10, 48, 49, 79] for the details. In fact, all the multipliers a satisfying this condition turn out to be of the form $a = i$ or $a = \lfloor m/i \rfloor$ for $i < \sqrt{m}$. In view of (29), it may be worthwhile considering negative multipliers a: it is possible that $-a > 0$ satisfies the condition (29) while $a + m$ (which is equivalent to a) does not. For small a, another approach which is often faster is to perform the computations in double-precision floating-point [48]. Techniques for computing $ax \bmod m$ in a high-level language for the more general case are studied by L'Ecuyer and Côté [52], who also give portable codes. A portable implementation of an MRG based on a characteristic trinomial with coefficients satisfying (29) is given in [51].

If $m = 2^e$ where e is the number of bits on the computer word, and if one can use unsigned integers without overflow checking, the products modulo m are easy to compute: just discard the overflow. This is quick and simple, and is the main reason why power of two moduli are still used in practice, despite their serious "statistical" drawbacks.

7.2. TAUSWORTHE, GFSR, AND TGFSR

Naïve software implementations of the digital multistep sequence (5)–(7) are rather slow in general, before s steps of the recurrence (5) must be performed for each u_n. However, very fast implementations are possible in some special cases. Hardware implementations are also possible via feedback shift registers.

For $m = 2$ (the Tausworthe generator), if one is willing to sacrifice memory for speed, then one can just implement the Tausworthe generator by implementing the equivalent GFSR generator, as explained in §3.6–3.7. But wasting that much memory could become a problem, especially when many parallel generators are required or helpful (see §8). Further, even if a GFSR is an acceptable option, one is probably better off with a TGFSR anyway. See Matsumoto and Kurita [66, 67] for how to implement the TGFSR.

Consider now a Tausworthe generator based on the characteristic trinomial $P(z) = z^k - z^{k-r} - 1$, and which satisfies $2r > k$ and $0 < s \leq r < k$. Define $q = k - r$. The following algorithm quickly computes s_n from s_{n-1}. Let A and B be bit vectors of size k and suppose that A initially contains $s_{n-1} = (x_{(n-1)s}, \ldots, x_{(n-1)s+k-1})$. The symbol $\oplus$ denotes the (bitwise) exclusive-or operator.

Algorithm 1.
1. $B \leftarrow q$-bit left-shift of A;
2. $B \leftarrow A \oplus B$;
3. $B \leftarrow (k - s)$-bit right-shift of B;
4. $A \leftarrow s$-bit left-shift of A;
5. $A \leftarrow A \oplus B$.

To simplify the notation in explaining how the algorithm works, assume (without loss of generality) that $n = 1$. Initially, A contains $(x_0, \ldots, x_{k-1})$. After step 2, the first r

bits of B contain $(x_0 \oplus x_q, \ldots, x_{r-1} \oplus x_{q+r-1}) = (x_k, \ldots, x_{k+r-1})$. After step 4, A contains $x_s, \ldots, x_{k-1}$ followed by s zeros, while B contains $k-s$ zeros followed by $x_k, \ldots, x_{k+s-1}$ (recall that $s \leq r$). Therefore, after step 5, A contains $(x_s, \ldots, x_{s+k-1})$. Then, A can be viewed as an unsigned integer and multiplied by the normalization constant 2^{-k} to produce u_n (here, $L = k$). If k is not larger than the computer's word size, this algorithm is fast and easy to program in any computer language which supports shifting and bitwise exclusive-or operations. Tezuka and L'Ecuyer [91] give an example in the C language. A FORTRAN code implementing a different algorithm, for the case $k = s$ (for which algorithm 1 does not work) is given in [10, p.216].

7.3. COMPLICATED CHARACTERISTIC POLYNOMIALS OR LARGE MODULI

Linear recurrences whose characteristic polynomials are a trinomial appear to allow much faster implementations than those based on polynomials with many non-zero coefficients, at least from what we saw so far. However, recurrences based on polynomials with few non-zero coefficients have important statistical defects [13, 59, 65, 66]. One way of getting around this problem is to combine two or more trinomial-based generators. Some classes of combined generators are in fact just efficient ways of implementing a recurrence whose characteristic polynomial has a large degree and many non-zero coefficients. This is the basic idea of the combined Tausworthe generators proposed in Tezuka and L'Ecuyer [91]. Their implementation turns out to be pretty fast, roughly as fast as that of a simple MLCG with prime modulus (depending on the computers and compilers), according to Tezuka [90]. Such combined generators are also recommended and studied in Wang and Compagner [97]. In a similar way, some MLCGs with very large moduli can be implemented efficiently via the combination of easily implementable MLCGs with small moduli. See §9 for further details. L'Ecuyer [48] gives examples of such implementations and explains how to do it in general. Other efficient ways of implementing MLCGs with large moduli are through the AWC and SWB generators discussed in §3.9. However, the latter generators always have a bad structure and must be avoided [15, 93].

7.4. RETURNING VECTORS OF RANDOM NUMBERS

James [41] observes that for fast generators, when the generator (procedure) returns one random number per call, the procedure call itself accounts for a large part of the time for generating the random number. He then recommends that each procedure call returns a *vector* of random numbers. Of course, if the size of the vector is large, this mechanism will be efficient only if all (or most) of the random numbers from the vector are used. This could speed up some simulation applications, but for the majority of applications, the savings will be negligible, while having to manage such a vector could be somewhat bothersome, especially to programmers who seek simplicity and elegance in their code. Anderson [6] gives FORTRAN codes to generate vectors of random numbers on vector computers.

8. Leapfrogging and Generating Numbers on Parallel Processors

There are two major situations which ask for generating (independent) multiple streams of random numbers in parallel:

(a) To perform a simulation on parallel processors, where each processor must generate its own random number stream [6];

(b) To assign different random number streams to different components of the model, for example to implement some variance reduction techniques, when performing a simulation on a single processor [10, 52].

Of course, these two situations can also be combined. To generate multiple streams in parallel, for either situation (a) or (b), the following approaches can be used:

(i) Use completely different generators for the different streams;

(ii) Use variants of the same generator; e.g., same modulus but different multipliers;

(iii) Use the same generator, but with different seeds.

Method (iii) is more convenient than (i) and (ii) in terms of management and implementation. Even when many good parameter sets are available, implementation considerations must be taken into account when selecting a generator. Often, the implementation of the selected generator is artfully crafted for speed and portability and some constants depending on the selected parameters must be precomputed for that purpose [48, 51, 91]. This tends to support approach (iii). Finding millions of good generators is not really a problem for some classes of generators like the LCG or MRG [21, 51], but not necessarily for all classes of generators. For example, for the two-component 32-bit combined Tausworthe generators proposed in [91], there is a limited number of good parameter sets. If many good parameter sets are available, one can conceivably maintain a large list of such good parameters to implement method (ii). These parameters must be computed beforehand and perhaps stored in a (permanent) file that would come with the simulation package. This seems more troublesome than approach (iii), which does not require storing that much information.

For the case of linear generators, the matrix approach (13) can be viewed as a way of formulating (iii). But in terms of speed, it is generally better to implement the corresponding MRG and run many copies of it in parallel.

Durst [21] suggests using (iii) with random seeds. Another approach is to select (deterministically) individual seeds that are far apart in the basic sequence. Typically, those seeds are evenly spaced and split the period of the generator into disjoint pieces, called *substreams*, long enough so that none of them could be exhausted in reasonable time. This is called *splitting* [52]. To generate the (far apart) seeds, for the case of a linear generator, just use the matrix formulation of the generator, with matrix multiplier A and modulus m. If X_n is the current seed, then $X_{n+\nu}$, for very large ν, can be computed directly as

$$X_{n+\nu} = (A^\nu \bmod m) X_n \bmod m.$$

The matrix $A^\nu \bmod m$ can be precomputed efficiently by a standard divide-to-conquer algorithm [49].

At first sight, splitting looks safer than generating seeds randomly. But one should be careful: it is a mined ground. The major concern is that of long range correlations, e.g., between X_n and $X_{n+\nu}$, X_n and $X_{n+2\nu}$, and so on. Extremely bad correlations occur, for example, when ν and the modulus m are both powers of two. This is why Durst [21] prefers random seeds. For further discussion on this and related topics, see [18, 21, 26, 49].

Niederreiter [76, 77] proposes different classes of *nonlinear* vector generators for use on parallel processors. Those generators appear interesting (at least theoretically), although specific (well tested) parameter values and efficient implementations are not given.

L'Ecuyer and Côté [52] have developed a random number package with two-level (imbedded) splitting facilities. It is based on the combined generator proposed in L'Ecuyer [48]. It provides for multiple generators running simultaneously, and each generator has its sequence of numbers partitioned into many long disjoint substreams. Simple procedure calls allow the user to make any generator jump ahead or backwards over those substreams. Similar packages could also be implemented rather easily using other (perhaps longer-period) generators.

9. Combined Generators

To increase the period, improve the statistical properties, and perhaps try to get rid of the lattice structure, different kinds of *combined* generators have been proposed. See [12, 41, 44, 49, 55, 60, 63, 64, 91, 97, 98] and other references given there. The structure of the hybrid generator thus obtained is often not well understood. Then, as pointed out by Ripley [83], using such generators may be a bit like playing ostrich. Theoretical results in [60, 11] appear to support the view (at first glance) that combined generators should have better statistical behavior in general than their individual components. However, as explained in [49], applying those theoretical results to "deterministic" generators is a somewhat shaky reasoning. Combination can conceivably worsen things. Nevertheless, empirical results strongly support combination [60, 50]. Most of the fast and simple generators (e.g., Tausworthe or MRGs based on primitive trinomials) happen to have statistical defects [13, 50, 59, 65]. Combining such fast generators could yield an efficient generator with much better statistical properties.

Recently some combined generators have been analyzed successfully and turn out to be equivalent, or approximately equivalent, to MLCGs with large (non-prime) moduli or to Tausworthe generators with large-degree (reducible) characteristic polynomials. Other classes of combined generators (like shuffling) are not (yet) well understood theoretically. See L'Ecuyer [49] and the references given there. We will now discuss two classes of combined generators which have been recently analyzed.

L'Ecuyer [48] proposed a combination method for MLCGs with distinct prime moduli

$m_1, \ldots, m_J$. If x_{jn} denotes the state of generator j at step n, define the combination:

$$Z_n = \left(\sum_{j=1}^{J} \delta_j x_{jn} \right) \mod m_1 \tag{30}$$

for some fixed integers δ_j. In [48], $\delta_j = (-1)^{j-1}$ and a specific generator are suggested. Wichmann and Hill [98] proposed a slightly different combination approach, which is a bit slower because it requires more divisions:

$$U_n = \left(\sum_{j=1}^{J} \delta_j x_{jn}/m_j \right) \mod 1. \tag{31}$$

If each individual MLCG has full period $m_j - 1$, then the period of the latter is always equal to the least common multiple of $m_1 - 1, \ldots, m_J - 1$ [49]. In practice, if the m_j's are distinct primes slightly smaller than 2^{31} and if the multipliers satisfy (29), then the generator is easy to implement on a 32-bit computer and can reach a very large period.

L'Ecuyer and Tezuka [55] have shown that there exists a MLCG with modulus $m = \prod_{j=1}^{J} m_j$ whose lattice structure approximates quite well the behavior of (30) in higher dimensions, and which is exactly equivalent to (31). This MLCG does not depend on the δ_j's. The equivalence of the Wichmann and Hill generator to a MLCG was already pointed out by Zeisel [98]. The results of [55] mean that (30) and (31) are almost equivalent. Such structural properties imply that the combined generators can be viewed as efficient ways of implementing MLCGs with very large moduli (with added "noise", in the case of (30)), which can be analyzed with the Beyer and spectral tests. Numerical and graphical illustrations are given in [55]. These combinations methods can also be generalized to the combination of MRGs with distinct prime moduli.

For the generator of L'Ecuyer [48], the lattice approximation is quite good in dimensions $t \geq 3$. As shown in [55], components with bad lattice structure can give rise to good combined generators, and the reverse is also true. Therefore, the selection of a combined generator should not be made just by selecting components with good lattice structure, as was done in [48], but by analyzing the lattice structure of the combined generator itself. Based on that criterion, better combined generators than the one proposed in [48] can be found [55]. Press and Teukolsky [80] propose a generator which adds a shuffle to the combined generator of [48]. That destroys the lattice structure.

Tezuka and L'Ecuyer [91] combine Tausworthe generators as follows. For each $j = 1, \ldots, J$, consider a Tausworthe generator with primitive characteristic polynomial $P_j(z)$ of degree k_j, with $s = s_j$ such that $\gcd(s_j, 2^{k_j} - 1) = 1$, and whose linear recurring sequence is $\{x_{j,n}, n \geq 0\}$. At step n, the output of generator j is produced by

$$u_{j,n} = \sum_{i=1}^{L} x_{j,ns+i-1} m^{-i}.$$

The output of the combined generator is defined as the bitwise exclusive-or of $u_{1,n}, \ldots, u_{J,n}$. If the polynomials $P_j(z)$ are pairwise relatively prime, then the period of the combined generator is the least common multiple of the individual periods, i.e., $\rho = \text{lcm}(2^{k_1} - 1, \ldots, 2^{k_J} - 1)$

(see [91]). Clearly, one should take distinct k_j's and the period ρ could then be the product of the individual periods. In that case, Wang and Compagner [97] call the sequence $\{x_n = x_{1,n} \oplus \cdots \oplus x_{J,n},\ n \geq 0\}$ an AM-sequence, where AM stands for "approximate maximum-length". As shown in [91, 97], the combined generator is equivalent to a Tausworthe generator with (reducible) characteristic polynomial $P(z) = P_1(z) \cdots P_J(z)$. The lattice structure and equidistribution properties of such combined generators are analyzed in [14, 91]. Tezuka and L'Ecuyer [91] suggest three specific combined generators, and give computer codes. TGFSRs can also be combined in a similar way and this is the subject of ongoing research.

One very attractive feature of that kind of combination is that even when the individual $P_j(z)$ have few non-zero coefficients (e.g., are trinomials) and bad statistical behavior, $P(z)$ often has many non-zero coefficients and the combined generator could be very good. In other words, this combination approach can be viewed as an efficient way of implementing Tausworthe generators with "good" characteristic polynomials. Compagner [13] and Wang and Compagner [97] also suggest the same kind of combination, and give supporting arguments. They show that the correlation structure of AM-sequences behaves very nicely in general. Their empirical investigation also suggests that when the number of non-zero coefficients in $P(z)$ is reasonably large, then the figure of merit $r^{(t)}$ defined in §5.3 is (usually) also large.

Marsaglia [60] recommends combining generators of different algebraic structures instead of combining generators within the same class. This is perhaps an interesting "scrambling" heuristic, but little theoretical analysis is available for such combinations. In [63, 64], he and his co-workers propose two specific combined generators of that sort. However, the generator of [63] has an important defect; as shown in [15], it has a lattice structure with distance d_t between hyperplanes of at least $1/\sqrt{6}$ for all $t \geq 45$.

10. Nonlinear Generators

Linear generators tend to have a too regular structure, and for that reason, many believe that the way to go is *nonlinear* [28, 29, 76, 77]. We can distinguish the following two ways of introducing nonlinearity in a generator:

(a) Use a generator with a linear transition function T, but transform the state nonlinearly to produce the output (G is nonlinear);

(b) Construct a generator with a nonlinear transition function T.

We will discuss one example of (a), namely the inversive congruential generator, and a few examples of (b). A common property of those nonlinear generators is that they do not produce a lattice structure like the linear ones. Their structure is highly nonlinear: typically, any t-dimensional hyperplane contains at most t overlapping t-tuples of successive values. Niederreiter [77] shows that they behave very much like truly random generators with respect to discrepancy. Therefore, their theoretical properties look quite good. However, specific well-tested parameter values with fast implementations are currently not available.

10.1. NONLINEAR CONGRUENTIAL GENERATORS OVER $\mathbb{F}_m$

Let $S = \mathbb{Z}_m$, where m is a large integer, and the output function be $G(x) = x/m$. Suppose that the transition function has the form

$$T(x) = f(x) \bmod m, \qquad x \in \mathbb{Z}_m. \tag{32}$$

This is an instance of case (b). Suppose now that m is prime and that f is selected so that the sequence $\{x_n\}$ defined by $x_n = T(x_{n-1})$, for any $x_0 \in \mathbb{Z}_m$, has (full) period $\rho = m$. Then, there exists a (unique) *permutation polynomial* P, of degree $k \leq m-2$, such that $P(n) = x_n$ for all $n \in \mathbb{Z}_m$ [76, 77]. Further, for all $t \leq k$, the "smallest" lattice which contains all the t-dimensional vectors of successive values $\boldsymbol{u}_n = (u_n, \ldots, u_{n+t-1})$ produced by the generator is the "complete" lattice $\mathbb{Z}^t/m$. In that case, i.e., when the vectors $\{\boldsymbol{u}_n\}$ span $\mathbb{Z}^t/m$ (over $\mathbb{Z}$), some authors say that the generator *passes the t-dimensional lattice test* [77]. Passing this test means that the points really do not have a lattice structure. This could be viewed as a desirable feature. So, the larger is k, the better. We must emphasize, however, that passing the lattice test does not mean at all that the generator has good statistical properties. The degree k of the polynomial P here also has to do with the discrepancy. It has been shown [76, 77] that for $2 \leq t \leq k$, $D_m^{(t)} \in O(km^{-1/2}(\log m)^t)$. For k close to m, this is the right order of magnitude.

10.2. INVERSIVE CONGRUENTIAL GENERATORS

Eichenauer et al. [22, 24, 25] introduced a class of nonlinear inversive generators which can be defined as the application of a supplementary step when transforming the x_n produced by an MRG into a value between 0 and 1 (and skipping the x_n's which are zero). Let $\{x_n\}$ be a full period linear recurring sequence (5), with prime m. Let $\tilde{x}_i$ be the i-th non-zero value x_n in that sequence. Define $z_n = (\tilde{x}_{n+1}\tilde{x}_n^{-1}) \bmod m$, where $\tilde{x}_n^{-1}$ is the inverse of $\tilde{x}_n$ in $\mathbb{F}_m$, and let the output be $u_n = z_n/m$. The inverse $\tilde{x}_n^{-1}$ can be computed via a version of Euclid's algorithm [44] or via $\tilde{x}_n^{-1} = x_n^{m-2} \bmod m$. Both methods to compute the inverse take time in $O(\log m)$ and have similar performance in practice. They are rather slow when implemented in software, which might make the generator unacceptably slow for certain applications. Fast hardware implementations are possible, though [22]. For prime m, the maximal possible period for $\{z_n\}$ is m^{k-1}. Sufficient conditions for it to be attained are given in [24, 28, 77]. Maximal period generators are easy to find. A nice property of inversive congruential generators with prime moduli is that their discrepancy is in the same order of magnitude as that of truly random numbers [28, 77].

For $k = 2$ or 3, one can also write a recursion directly for z_n. For $k = 2$, it is

$$z_n = \begin{cases} (a_1 + a_2 z_{n-1}^{-1}) \bmod m & \text{if } z_{n-1} \neq 0; \\ a_1 & \text{if } z_{n-1} = 0. \end{cases}$$

In that case, a sufficient condition for maximal period is that $P(z) = z^2 - a_1 z - a_2$ is a primitive polynomial over $\mathbb{F}_m$.

A slightly different variant is the *explicit inversive congruential* method, introduced by Eichenauer-Herrmann [27]. Here, $x_n = an + c$, for $n \geq 0$, where $a \neq 0$ and c are in $\mathbb{Z}_m$, $z_n = x_n^{-1} = (an + c)^{m-2} \bmod m$, and $u_n = z_n/m$. The period is $\rho = m$ and the permutation polynomial P associated with this generator has degree $k = m - 2$. So, the generator "passes the lattice test" in all dimensions $t \leq m - 2$. Niederreiter [76, 78] also shows that every hyperplane in $\mathbb{R}^t$ contains at most t points from the set $\{\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{m-1}\}$, and obtains discrepancy bounds.

Inversive congruential generators with power-of-two moduli have also been studied [25, 28, 29]. For these generators, non-trivial upper bounds on the discrepancy are available only in dimension 2. Further, the generated points have some regular structures [28]. Therefore, prime moduli appear preferable.

Inversive congruential generators have been the subject of many papers in the last five years or so. Eichenauer-Herrmann [28] gives a survey, as well as a few suggested parameters. These generators would perhaps deserve a more extensive coverage than what is done here. The present coverage reflects the relative lack of practical experience of the author with their use. They certainly deserve further investigation.

10.3. QUADRATIC CONGRUENTIAL GENERATORS

A special case which has received some attention is the *quadratic* case, for which f in (32) has the quadratic form $f(x) = ax^2 + bx + c$, with $a, b, c \in \mathbb{Z}_m$ [44, 77]. If m is a power of two, then the generator has full period ($\rho = m$) if and only if a is even, $(b - a) \bmod 4 = 1$, and c is odd [44]. The points produced by that generator turn out to lie on a union of grids, which can be determined explicitly [23]. Bounds on $D_m^{(t)}$ are given in [77].

10.4. OTHER NONLINEAR GENERATORS

Some nonlinear generators have also been proposed by people from the field of cryptology [8, 54, 85]. Blum, Blum, and Shub [8] proposed the following class, known as BBS generators. Let $m = pq$ be a *Blum integer*, i.e., such that p, q are two distinct primes both congruent to 3 modulo 4. Let $x_0 = x^2 \bmod m$, where x is a positive integer such that $\gcd(x, m) = 1$, and for $n \geq 1$, let

$$x_n = x_{n-1}^2 \bmod m.$$

At each step, the generator outputs the last ν bits of x_n. Suppose that both p and q are $k/2$-bit integers and that $\nu \in O(\log k)$. Under the reasonable assumption that factoring Blum integers is hard, it has been proved that no polynomial-time (in k) statistical test can distinguish (in some specific sense) a BBS generator from a truly random one. This means that for large enough k, the generator should behave very nicely from a statistical point of view. See [8, 54] for further details. However, L'Ecuyer and Proulx [54] show that a software implementation of the BBS generator is much too slow for practical use in simulation applications. More efficient generators with the same kind of polynomial-time statistical "perfectness" have been proposed recently [85]. Further investigation is required before assessing their practical competitivity for simulation.

11. Empirical Statistical Testing

An unlimited number of empirical tests can be designed for random number generators. The null hypothesis is H_0: "The sequence is a sample of i.i.d. $U(0,1)$ random variables", and a statistical test tries to find empirical evidence against H_0 (usually, against unspecified alternatives). Any function of a finite number of $U(0,1)$ random variables, whose (sometimes approximate) distribution under H_0 is known, can be used as a statistic T which defines a test for H_0.

11.1. MULTILEVEL TESTS

To increase the power, a given test can be replicated N times, on disjoint parts of the sequence, yielding values $T_1, \ldots, T_N$ of the statistic. The empirical distribution of those N values can then be compared to the theoretical distribution of T under H_0, via standard univariate goodness-of-fit tests, like Kolmogorov-Smirnov (KS), Anderson-Darling, or Cramer-von Mises [87]. We call this a *two-level* test.

For example, one can compute the value d of the KS statistic D_N, and the descriptive level δ_2 of the two-level test, defined as

$$\delta_2 = P[D_N > d \mid H_0]. \tag{33}$$

Under H_0, δ_2 should be $U(0,1)$. A very small value of δ_2 (say, $\delta_2 < .01$) provides evidence against H_0. In case of doubt, the whole procedure can be repeated (independently), and if small values of δ_2 are produced consistently, H_0 should be rejected, which means that the generator *fails* the test. If δ_2 is not too small, that improves confidence in the generator. It should be clear, however, that statistical tests never *prove* that a generator is foolproof.

Here, we did not specify an alternative to H_0. So, the power of the test is not really a well-defined notion. Empirically, however, two-level testing tends to catch up more easily the defective generators, at least for the current "standard" tests, and this is what we mean by "increasing the power".

One can also perform a *three-level* test: replicate the two-level test R times, and compare the empirical distribution of the R values of δ_2 with the $U(0,1)$ distribution, using again a goodness-of-fit test, and yielding a descriptive level δ_3. Reject if δ_3 is too small. This can be taken up to fourth level, fifth level, and so on. However, one major problem with higher-level tests is that in most cases, the exact distribution of the first-level statistic T under H_0 is not available, but only an approximation of it is (e.g., a chi-squared distribution). Often, that approximation is also the asymptotic distribution as the (first-level) sample size increases to infinity. Higher-level tests may then detect the lack-of-fit of that approximation long before detecting any problem with the generator. Good generators will then be rejected. The higher the level, the more this is likely to happen. Perhaps this problem could be alleviated in some cases by finding better statistics or better approximations, but is usually hard to eliminate. Similarly, for a three-level (or more) test, if a KS statistic is used at the second level, the N descriptive level values δ_2 will usually be computed using the asymptotic (as $N \to \infty$) KS distribution. Again, if N is too small, the test will detect the fact that the asymptotic is not yet a good enough approximation. So, for higher-level tests, one must take much larger sample sizes at the lower levels. This quickly becomes time-wise prohibitive.

If higher-level tests are problematic, why not just use one level ? One-level tests do not test the local behavior of generators as well as higher-level tests. Some sequences have good properties when we take the average over the whole sequence, but not when we look at very short subsequences. As an illustration, consider the (extreme) example of a generator producing the values $i/2^{31}$, $i = 1, 2, \ldots, 2^{31} - 1$, in that order. A uniformity test over the whole sequence will give a perfect adjustment. In fact, the adjustment will be too good, giving what is called *super-uniformity* [87]. On the other hand, uniformity tests over disjoint shorter subsequences will give terribly bad adjustments. So, one-level tests are not always appropriate, and two-level tests seem to offer a good compromise.

11.2. STANDARD AND MORE STRINGENT TESTS

Knuth [44] describes a set of tests which have been considered for a while as "the standard tests" for testing random number generators. Arguing that those so-called standard tests were not sufficiently discriminatory, i.e., that many "bad" generators passed most of the tests, Marsaglia [60] proposed a new set of more stringent tests. Indeed, sophisticated applications like probabilistic computational geometry, probabilistic combinatorial algorithms, design of statistical tests, and so on, often require generators with excellent high-dimensional structures. Marsaglia argued that for such classes of applications, simple generators (e.g., LCG, Tausworthe, GFSR, etc.) were not good enough, and advocated combined generators. Other statistical tests for random number generators are proposed or discussed in [17, 31, 43, 50, 66, 68, 87] and the references given there.

For many interesting statistical tests, the theoretical distribution of the associated statistic is unfortunately unknown, at least in practically usable form [50, 60]. In such situations, Marsaglia [60] suggests comparing the empirical distribution of a generator to be tested with that of a "good" generator. But which generator should we use for that ? We get into a vicious circle, because what we want to test is precisely whether the random number generators are able to reproduce the right distribution function for T. In practice, though, estimating the theoretical distribution with many different types of (supposedly good) random number generators could be a reasonable (heuristic) compromise. If the results agree, it will certainly improve our confidence that this is the right distribution.

11.3. EXAMPLES OF TESTS RESULTS

L'Ecuyer [50] has applied 10 (two-level) statistical tests to 8 popular or recently proposed random number generators. The tests included the poker test, the runs-up test, the birthday spacings test, OPSO (with four different sets of parameters), and the nearest pair test (in dimensions 4, 6, and 9). See [50] for more details. The generators were the first 8 listed in Table 1. The author also applied the same tests to the MRG defined by

$x_n = (107374182 x_{n-1} + 104480 x_{n-5}) \bmod m$; $u_n = x_n/m$, with $m = 2^{31} - 1$, taken from [51] (G9 in the table).

Table 1: The generators tested.

G1.	MLCG with $m = 2^{31} - 1$ and $a = 16807$.
G2.	MLCG with $m = 2^{31} - 1$ and $a = 630360016$.
G3.	MLCG with $m = 2^{31} - 1$ and $a = 742938285$.
G4.	CSD generator of Sherif and Dear [86].
G5.	Combined generator in Fig. 3 of L'Ecuyer [48].
G6.	Combined Tausworthe generator G1 of Tezuka and L'Ecuyer [91].
G7.	Twisted GFSR with $(r, s, p) = (25, 7, 32)$.
G8.	Subtract-with-borrow generator with $(b, r, s, L) = (2^{32} - 5, 43, 22, 1)$.
G9.	MRG with $m = 2^{31} - 1$, $k = 5$ and $a_1 = 107374182$, $a_5 = 104480$, $a_2 = a_3 = a_4 = 0$

Generators G1 and G2 are used in various software packages [10, 47] and recommended by some authors [79, 47]. Fishman and Moore [33] recommend G3. G7 is proposed by Matsumoto and Kurita [66], while G8 is proposed by Marsaglia and Zaman [62] and further recommended by James [41]. The results were that besides G5, G6, and G9, all other generators failed spectacularly at least one of the tests. Moreover, each of G1 to G4 failed spectacularly at least 6 tests out of 10. In more than half of the "fail" cases, the descriptive level δ_2 was less than 10^{-15}. Clearly, these results should be shocking to many simulation practitioners.

Some of the test results could be explained by looking at the structure of the generator. For example, as pointed out by Ripley [81, 82], the MLCGs are bound to fail the nearest-pair test because of their regular lattice structure: the length of the shortest vector in the Minkowski reduced basis is a lower bound on the distance between points, so that the nearest pair in a large set of points cannot be as close as it should be statistically. A nearest-pair test can also be constructed using the norm on the space of formal series defined in §5.2. Tausworthe and GFSR generators with good equidistribution properties are likely to fail such tests (see [89]). Other examples of tests that certain classes of generators are bound to fail are given in [59, 67]. Results of extensive statistical tests can also be found in [43, 48, 60]. In [43], the authors have applied a battery of tests to 35 generators proposed in the literature, and recommend three generators on the basis of their speed and their performance in those tests. Two of those generators are LCGs with modulus 2^{32}, while the third one is a combined generator. The two recommended LCGs would certainly fail some of the 10 tests applied in [50], for the reasons explained above.

11.4. WHICH TESTS ARE THE GOOD ONES ?

Statistical tests are far from being clean-cut testing tools. Because any generator has finite period, almost any good test, if run long enough, will eventually detect regularity and reject the generator. So, how can we be satisfied with empirical test results ? A reasonable practical view here is to restrict ourselves to tests that we can practically run on a computer. For example, if a test needs 10 million years of CPU time on the world's fastest computer, perhaps we do not care much about its eventual results. However, we would like the generator to pass (with probability close to one) all known tests which can run in, say, less than a few hours, assuming that the generator's structure is unknown to the test builder and only the output values u_n are observed. But even this is not easy to achieve with efficient generators. If the generator is a MLCG, for example, there exist efficient algorithms which can find out the modulus and multiplier only from the output values, and guess the next values [9, 45]. From that, it is easy to design a test that the MLCG will fail. Perhaps asking a generator to pass all such tests is asking too much ? Well, that depends on the application. If a generator fails a given statistical test, it is easy, from that, to construct an application for which the generator will produce completely wrong results. One trivial example: Suppose you want to estimate the distribution of the statistic T on which the test is based. If the generator fails the test, it means that the distribution of T is not correctly estimated.

Acknowledgements

This work has been supported by NSERC-Canada grant # OGP0110050 and FCAR-Québec grant # 93ER1654. The paper was written while the author was a Visiting Professor, under the Toshiba Chair, in the Graduate School of Science and Engineering, at Waseda University, Tokyo, Japan. Most of the text was written in the gardens of various temples surrounding Kyoto. I wish to thank Raymond Couture, Bennett L. Fox, Makoto Matsumoto, Harald Niederreiter, and Shu Tezuka for their valuable comments.

References

[1] L. Afflerbach, The Sub-Lattice Structure of Linear Congruential Random Number Generators, *Manuscripta Math.* **55** (1986) 455–465.

[2] L. Afflerbach and H. Grothe, Calculation of Minkowski-Reduced Lattice Bases. *Computing* **35** (1985) 269–276.

[3] L. Afflerbach and H. Grothe, The Lattice Structure of Pseudo-Random Vectors Generated by Matrix Generators. *J. of Computational and Applied Math.* **23** (1988) 127–131.

[4] L. Afflerbach and R. Weilbächer, The Exact Determination of Rectangle Discrepancy for Linear Congruential Pseudorandom Numbers, *Math. of Computation* **53**, 187 (1989) 343–354.

[5] D. L. André, G. L. Mullen, and H. Niederreiter, Figures of Merit for Digital Multistep Pseudorandom Numbers, *Math. of Computation* **54** (1990) 737–748.

[6] S. L. Anderson, Random Number Generators on Vector Supercomputers and Other Advanced Architectures, *SIAM Review* **32** (1990) 221–251.

[7] A. C. Atkinson, Tests of Pseudo-Random Numbers, *Applied Statistics* **29** (1980) 164–171.

[8] L. Blum, M. Blum, and M. Shub, A Simple Unpredictable Pseudo-Random Number Generator, *SIAM J. Comput.* **15**, 2 (1986) 364–383.

[9] J. Boyar, Inferring Sequences Produced by a Linear Congruential Generator Missing Low-Order Bits, *Journal of Cryptology* **1** (1989) 177–184.

[10] P. Bratley, B. L. Fox, and L. E. Schrage, *A Guide to Simulation*, second edition. Springer-Verlag, New York (1987).

[11] M. Brown and H. Solomon, On Combining Pseudorandom Number Generators, *Annals of Statistics* **1** (1979) 691–695.

[12] B. J. Collings, Compound Random Number Generators, *J. of the American Statistical Association* **82**, 398 (1987) 525–527.

[13] A. Compagner, The Hierarchy of Correlations in Random Binary Sequences, *Journal of Statistical Physics* **63** (1991) 883–896.

[14] R. Couture, P. L'Ecuyer, and S. Tezuka, On the Distribution of k-Dimensional Vectors for Simple and Combined Tausworthe Sequences, *Mathematics of Computation* **60** (1993) To appear.

[15] R. Couture and P. L'Ecuyer, On the Lattice Structure of Certain Linear Congruential Sequences Related to AWC/SWB Generators, *Mathematics of Computation*, To appear.

[16] J. Dagpunar, *Principles of Random Variate Generation*, Oxford University Press, 1988.

[17] J. W. Dalle Molle, M. J. Hinich, and D. J. Morrice, Higher-Order Cumulant Spectral Based Statistical Tests of Pseudo Random Variate Generators, *Proceedings of the 1992 Winter Simulation Conference*, IEEE Press (1992) 618–625.

[18] A. De Matteis and S. Pagnutti, Parallelization of Random Number Generators and Long-Range Correlations, *Numerische Mathematik* **53** (1988) 595–608.

[19] L. Devroye, *Non-Uniform Random Variate Generation*, Springer-Verlag, New York (1986).

[20] E. J. Dudewicz and T. G. Ralley, *The Handbook of Random Number Generation and Testing with TESTRAND Computer Code*, American Sciences Press, Columbus, Ohio (1981).

[21] M. J. Durst, Using linear congruential generators for parallel random number generation, *Proceedings of the 1989 Winter Simulation Conference*, IEEE Press (1989) 462–466.

[22] J. Eichenauer and J. Lehn, A Nonlinear Congruential Pseudorandom Number Generator, *Statistische Hefte* **27** (1986) 315–326.

[23] J. Eichenauer and J. Lehn, On the Structure of Quadratic Congruential Sequences, *Manuscripta Math.* **58** (1987) 129–140.

[24] J. Eichenauer, H. Grothe, J. Lehn, and A. Topuzoğlu, A Multiple Recursive Nonlinear Congruential Pseudorandom Number Generator, *Manuscripta Math.* **59** (1987) 331–346.

[25] J. Eichenauer, J. Lehn, and A. Topuzoğlu, A Nonlinear Congruential Pseudorandom Number Generator with Power of Two Modulus, *Math. of Computation* **51**, 184 (1988) 757–759.

[26] J. Eichenauer-Herrmann, A Remark on Long-Range Correlations in Multiplicative Congruential Pseudo Random Number Generators, *Numerische Mathematik* **56** (1989) 609–611.

[27] J. Eichenauer-Herrmann, Statistical Independence of a New Class of Inversive Congruential Pseudorandom Numbers, *Mathematics of Computation* **60** (1993) 375–384.

[28] J. Eichenauer-Herrmann, Inversive Congruential Pseudorandom Numbers: a Tutorial, *International Statistical Reviews* **60** (1992) 167–176.

[29] J. Eichenauer-Herrmann and H. Grothe, A New Inversive Congruential Pseudorandom Number Generator with Power of Two Modulus, *ACM Transactions of Modeling and Computer Simulation* **2**, 1 (1992) 1–11.

[30] J. Eichenauer-Herrmann, H. Grothe, and J. Lehn, On the Period Length of Pseudorandom Vector Sequences Generated by Matrix Generators, *Math. of Computation* **52**, 185 (1989) 145–148.

[31] E. D. Erdmann, Empirical Tests of Binary Keystreams. Master's thesis, Department of Mathematics, Royal Holloway and Bedford New College, University of London, 1992.

[32] A. M. Ferrenberg, D. P. Landau, and Y. J. Wong, Monte Carlo Simulations: Hidden Errors from "Good" Random Number Generators, *Physical Review Letters* **69**, 23 (1992) 3382–3384.

[33] G. S. Fishman and L. S. Moore III, An Exhaustive Analysis of Multiplicative Congruential Random Number Generators with Modulus $2^{31} - 1$, *SIAM J. on Scientific and Statistical Computing* **7**, 1 (1986) 24–45.

[34] G. S. Fishman, Multiplicative Congruential Random Number Generators with Modulus 2^β: An Exhaustive Analysis for $\beta = 32$ and a Partial Analysis for $\beta = 48$, *Mathematics of Computation* **54**, 189 (Jan 1990) 331–344.

[35] M. Fushimi, An Equivalence Relation between Tausworthe and GFSR Sequences and Applications, *Applied Math. Letters* **2**, 2 (1989) 135–137.

[36] M. Fushimi and S. Tezuka, The k-Distribution of Generalized Feedback Shift Register Pseudorandom Numbers, *Communications of the ACM* **26**, 7 (1983) 516–523.

[37] H. Grothe, Matrix Generators for Pseudo-Random Vectors Generation, *Statist. Hefte* **28** (1987) 233–238.

[38] H. Grothe, Matrixgeneratoren zur Erzeugung gleichverteilter Pseudozufallsvektoren (in german), Dissertation (thesis), Tech. Hochschule Darmstadt, Germany, 1988.

[39] J. R. Heringa, H. W. J. Blöte, and A. Compagner, New Primitive Trinomials of Mersenne-Exponent Degrees for Random-Number Generation, *International Journal of Modern Physics C* **3**, 3 (1992) 561–564.

[40] D. C. Hoaglin and M. L. King, A Remark on Algorithm AS 98: The Spectral Test for the Evaluation of Congruential Pseudo-random Generators, *Applied Statistics* **27** (1978) 375–377.

[41] F. James, A review of pseudorandom number generators, *Computer Physics Communications*, **60** (1990) 329-344.

[42] R. Kannan, A. K. Lenstra, and L. Lovász, Polynomial Factorization and Nonrandomness of Bits of Algebraic and Some Transcendental Numbers, *Math. of Computation*, **50**, 181 (1988) 235–250.

[43] Z. A. Karian and E. J. Dudewicz, *Modern Statistical, Systems, and GPSS Simulation: The First Course*, Computer Science Press, Freeman, New York, 1991.

[44] D. E. Knuth, *The Art of Computer Programming : Seminumerical Algorithms*, vol. 2, second edition. Addison-Wesley, 1981.

[45] H. Krawczyk, How to Predict Congruential Generators, in *Lecture Notes in Computer Science 435; Advances in Cryptology: Proceedings of Crypto'89*, G. Brassard, Ed., Springer-Verlag, Berlin (1990) 138–153.

[46] Y. Kurita and M. Matsumoto, Primitive t-nomials ($t = 3, 5$) over $GF(2)$ whose Degree is a Mersenne Exponent ≤ 44497, *Mathematics of Computation* **56**, 194 (1991) 817–821.

[47] A. M. Law and W. D. Kelton, *Simulation Modeling and Analysis*, Second edition, McGraw-Hill (1991).

[48] P. L'Ecuyer, Efficient and Portable Combined Random Number Generators. *Communications of the ACM* **31**, 6 (1988) 742–749 and 774. See also the correspondence in the same journal, **32**, 8 (1989) 1019–1024.

[49] P. L'Ecuyer, Random Numbers for Simulation, *Communications of the ACM* **33**, 10 (1990) 85–97.

[50] P. L'Ecuyer, Testing Random Number Generators, *Proceedings of the 1992 Winter Simulation Conference*, IEEE Press (1992), 305–313.

[51] P. L'Ecuyer, F. Blouin, and R. Couture, A Search for Good Multiple Recursive Random Number Generators, *ACM Transactions on Modeling and Computer Simulation* **3**, 2 (1993) 87–98.

[52] P. L'Ecuyer and S. Côté, Implementing A random number package with splitting facilities, *ACM Trans. on Math. Software* **17** (1991) 98–111.

[53] P. L'Ecuyer and R. Couture, An Implementation of the Lattice and Spectral Tests for Linear Congruential and Multiple Recursive Generators, In preparation.

[54] P. L'Ecuyer and R. Proulx, About Polynomial-Time "Unpredictable" Generators. *Proceedings of the 1989 Winter Simulation Conference*, IEEE Press (1989) 467–476.

[55] P. L'Ecuyer and S. Tezuka, Structural Properties for Two Classes of Combined Random Number Generators, *Mathematics of Computation* **57**, 196 (1991) 735–746.

[56] A. K. Lenstra, Factoring Multivariate Polynomials over Finite Fields, *J. Comput. Syst. Science* **30** (1985) 235–248.

[57] T. G. Lewis and W. H. Payne, Generalized Feedback Shift Register Pseudorandom Number Algorithm, *J. of the ACM* **20**, 3 (1973) 456–468.

[58] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and Their Applications*, Cambridge University Press, Cambridge (1986).

[59] J. H. Lindholm, An Analysis of the Pseudo-Randomness Properties of Subsequences of Long m-Sequences, *IEEE Transactions on Information Theory* **IT-14**, 4 (1968) 569–576.

[60] G. Marsaglia, A Current View of Random Number Generation, *Computer Science and Statistics, Proceedings of the Sixteenth Symposium on the Interface*, Elsevier Science Publ. (North-Holland) (1985) 3–10.

[61] G. Marsaglia and L.-H. Tsay, Matrices and the Structure of Random Number Sequences, *Linear Algebra and its Applications* **67** (1985) 147–156.

[62] G. Marsaglia and A. Zaman, A New Class of Random Number Generators, *The Annals of Applied Probability* **1** (1991) 462–480.

[63] G. Marsaglia, B. Narasimhan, and A. Zaman, A Random Number Generator for PC's, *Computer Physics Communications* **60** (1990) 345–349.

[64] G. Marsaglia, A., Zaman, and W. W. Tsang, Towards a Universal Random Number Generator, *Stat. and Prob. Letters* **8** (1990) 35–39.

[65] M. Matsumoto and Y. Kurita, The Fixed Point of an m-sequence and Local Non-Randomness, technical report 88-027, Department of Information Science, University of Tokyo (1988).

[66] M. Matsumoto and Y. Kurita, Twisted GFSR Generators, *ACM Transactions on Modeling and Computer Simulation* **2**, 3 (1992) 179–194.

[67] M. Matsumoto and Y. Kurita, Well-Tempered TGFSR Generators, Manuscript (1992).

[68] U. M. Maurer, A Universal Statistical Test for Random Bit Generators, *Journal of Cryptology* **5** (1992) 89–105.

[69] H. Niederreiter, Quasi-Monte Carlo Methods and Pseudorandom Numbers, *Bull. Amer. Math. Soc.* **84**, 6 (1978) 957–1041.

[70] H. Niederreiter, The Serial Test for Pseudorandom Numbers Generated by the Linear Congruential Method, *Numer. Math.* **46** (1985), 51–68.

[71] H. Niederreiter, A Pseudorandom Vector Generator Based on Finite Field Arithmetic, *Math. Japonica* **31** (1986) 759–774.

[72] H. Niederreiter, A Statistical Analysis of Generalized Feedback Shift Register Pseudorandom Number Generators, *SIAM J. Sci. Stat. Comput.* **8** (1987) 1035–1051.

[73] H. Niederreiter, The Serial Test for Digital k-Step Pseudorandom Numbers, *Mathematical Journal of Okayama University* **30** (1988) 93–119.

[74] H. Niederreiter, Statistical Independence Properties of Pseudorandom Vectors Produced by Matrix Generators, *J. Comput. Appl. Math.* **31** (1990) 139–151.

[75] H. Niederreiter, Recent Trends in Random Number and Random Vector Generation, *Annals of Operations Research* **31** (1991) 323–345.

[76] H. Niederreiter, New Methods for Pseudorandom Number and Pseudorandom Vector Generation, *Proceedings of the 1992 Winter Simulation Conference*, IEEE Press (1992) 264–269.

[77] H. Niederreiter, *Random Number Generation and Quasi-Monte Carlo Methods*, SIAM CBMS-NSF Regional Conference Series in Applied Mathematics, vol. 63, SIAM, Philadelphia (1992).

[78] H. Niederreiter, On a New Class of Pseudorandom Numbers for Simulation Methods, *J. of Computational and Applied Math.*, To appear.

[79] S. K. Park and K. W. Miller, Random Number Generators: Good Ones Are Hard to Find, *Communications of the ACM* **31**, 10 (1988) 1192–1201.

[80] W. H. Press and S. A. Teukolsky, Portable Random Number Generators, *Computers in Physics* **6**, 5 (1992) 522–524.

[81] B. D. Ripley, The Lattice Structure of Pseudo-random Number Generators, *Proc. Roy. Soc. London, Series A* **389** (1983) 197–204.

[82] B. D. Ripley, *Stochastic Simulation*, Wiley, New York (1987).

[83] B. D. Ripley, Uses and Abuses of Statistical Simulation, *Mathematical Programming* **42** (1988) 53–68.

[84] B. D. Ripley, Thoughts on Pseudorandom Number Generators, *J. of Computational and Applied Mathematics* **31** (1990) 153–163.

[85] A. W. Schrift and A. Shamir, The Discrete Log is Very Discreet, *Proceedings of STOC'90*, ACM Publications (1990) 405–415.

[86] Y. S. Sherif and R. G. Dear, Development of a New Composite Pseudo-Random Number Generator, *Microelectronics and Reliability* **30** (1990) 545–553.

[87] M. S. Stephens, Tests for the Uniform Distribution, in *Goodness-of-Fit Techniques*, Edited by R. B. D'Agostino and M. S. Stephens, Marcel Dekker (1986) 331–366.

[88] R. C. Tausworthe, Random Numbers Generated by Linear Recurrence Modulo Two, *Math. of Computation* **19** (1965) 201–209.

[89] S. Tezuka, Lattice Structure of Pseudorandom Sequences From Shift-Register Generators, *Proceedings of the 1990 Winter Simulation Conference*, IEEE Press (1990) 266–269.

[90] S. Tezuka, A Unified View of Long-Period Random Number Generators, Submitted for publication (1992).

[91] S. Tezuka and P. L'Ecuyer, Efficient and Portable Combined Tausworthe Random Number Generators. *ACM Transactions on Modeling and Computer Simulation* **1** (1991) 99–112.

[92] S. Tezuka and P. L'Ecuyer, Analysis of Add-with-Carry and Subtract-with-Borrow Generators, *Proceedings of the 1992 Winter Simulation Conference*, IEEE Press (1992) 443–447.

[93] S. Tezuka, P. L'Ecuyer, and R. Couture, On the Lattice Structure of the Add-with-Carry and Subtract-with-Borrow Random Number Generators, *ACM Transactions of Modeling and Computer Simulation*, To appear.

[94] S. Tezuka and M. Fushimi, Calculation of Fibonacci Polynomials for GFSR Sequences with Low Discrepancies, *Mathematics of Computation* **60** (1993) To appear.

[95] J. P. R. Tootill, W. D. Robinson, and A. G. Adams, The Runs up-and-down Performance of Tausworthe Pseudo-Random Number Generators, *J. of the ACM* **18** (1971) 381–399.

[96] J. P. R. Tootill, W. D. Robinson, and D. J. Eagle, An Asymptotically Random Tausworthe Sequence, *J. of the ACM* **20** (1973) 469–481.

[97] D. Wang and A. Compagner, On the Use of Reducible Polynomials as Random Number Generators, *Mathematics of Computation*, **60** (1993) 363–374.

[98] B. A. Wichmann and I. D. Hill, An Efficient and Portable Pseudo-random Number Generator. *Applied Statistics* **31** (1982) 188–190. See also corrections and remarks in the same journal by Wichmann and Hill **33** (1984) 123; McLeod **34** (1985) 198–200; Zeisel **35** (1986) 89.