SERIAL AND PARALLEL RANDOM NUMBER GENERATION

MICHAEL MASCAGNI
Doctoral Program in Scientific Computing **and**
Department of Mathematics
Box 10057
University of Southern Mississippi
Hattiesburg, MS 39406-0057
e-mail: Michael.Mascagni@usm.edu

Abstract. We detail several methods used in the production of computational random numbers. These include pseudorandom numbers, which mimic the behavior of real random numbers, and quasirandom numbers, which provide very uniformly distributed sets of numbers. We first discuss methods of quasirandom number generation . Quasirandom numbers, numbers that are highly uniformly distributed, are preferred in Monte Carlo applications where an even sampling of the computational space is more important than randomness. We will then give an introduction to various methods for pseudorandom number generation. We will do this in a rather specialized way by presenting pseudorandom number generation methods that are suitable for parallel use. We will focus on methods based on parameterization, meaning that we will not consider splitting methods. We describe parameterized versions of the following pseudorandom number generators: (i) linear congruential generators, (ii) linear matrix generators, (iii) shift-register generators, (iv) lagged-Fibonacci generators, and (v) inversive congruential generators.

We briefly describe the methods, detail some advantages and disadvantages of each method and recount results from number theory that impact our understanding of their quality in parallel applications. Finally, we present a short description of a scalable library for pseudorandom number generation, called SPRNG. The description contained within this document is meant only to outline the rationale behind and the capabilities of SPRNG. Much more information, including examples and detailed documentation aimed at helping users with putting and using SPRNG on scalable systems is available at the URL: www.ncsa.edu/Apps/SPRNG.

1. Introduction

Monte Carlo applications are widely perceived as embarrassingly parallel.[1] The truth of this notion depends, to a large extent, on the quality of the parallel random number generators used. It is widely assumed that with N processors executing N copies of a Monte Carlo calculation, the pooled result will achieve a variance N times smaller than a single instance of this calculation in the same amount of time. This is true only if the results in each processor are statistically independent. In turn, this will be true only if the streams of random numbers generated in each processor are independent.

Parallelism, as described above, is one approach to accelerate convergence of Monte Carlo computations. Another is to use quasirandom numbers instead of pseudorandom numbers. Quasirandom numbers take advantage of the fact many applications do not need randomness but uniformity from their random number generator. Quasirandom numbers are highly uniform in distribution, and have significantly superior convergence over pseudorandom numbers for applications that require uniformity. Numerical Integration by Monte Carlo is the classic example of an application that in reality requires uniformity, not randomness. With pseudorandom numbers, N samples reduce the stochastic errors by $O(N^{-1/2})$, while quasi-Monte Carlo methods can produce *deterministic* errors of $O(N^{-1}(\log N)^{s-1})$ in s-dimensional quadrature. Quasirandom numbers have also been used to accelerate the Monte Carlo convergence of other application, and so are sought by computational scientists. To date, quasirandom numbers have only been developed in a serial computational context. Thus we begin our exposition of modern random number generation methods with a brief introduction to quasirandom numbers. Then we briefly present several methods for parallel pseudorandom number generation and discuss pros and cons for each method. If the reader is interested in background material on plain old serial pseudorandom number generation in general, consult the following references: [21], [25], [47], [49], while a good overview of parallel pseudorandom number generation can be found in [38].

In our parallel pseudorandom number generation review we are interested, exclusively, with methods for obtaining **p**arallel **p**seudorandom **n**umber **g**enerator**s** (PPRNGs) via parameterization. The exact meaning of parameterization depends on the type of PRNG under discussion, but we wish to distinguish parameterization from splitting methods. We will not be considering the production of parallel streams of pseudorandom num-

[1] Monte Carlo enthusiasts prefer the term "naturally parallel" to the somewhat derogatory "embarrassingly parallel" coined by computer scientists.

bers by taking substreams from a single, long-period PRNG. For readers interested in splitting methods and the consequences of using split streams in parallel please consult: [9], [11], [12], [13], [18], [26]. In general, we seek to determine a parameter in the underlying recursion of the PRNG that can be varied. Each valid value of this parameter will lead to a recursion that produces a unique, full-period stream of pseudorandom numbers. We then discuss efficient means to specify valid parameter values and consider these choices in terms of the quality of the pseudorandom numbers produced.

The plan of the paper is as follows. We begin our review with a brief, self-contained introduction to quasirandom number generation in §2. In §3 we present an extensive overview of parallel pseudorandom number generation mostly viewed from the parameterization point of view. In §3.1 two methods for parameterizing **l**inear **c**ongruential **g**enerator**s** (LCGs). In §3.2 we briefly present and discuss linear matrix methods for parallel pseudorandom number generation. In §3.3 we present a parameterization of another linear method: **s**hift-**r**egister **g**enerator**s** (SRGs). This parameterization is analogous to one of the LCG parameterizations presented in §3.1. In §3.4 we consider the parallel parameterization of so-called lagged-Fibonacci generators, and in §3.5 we present a parallel nonlinear random number generator called the **i**nversive **c**ongruential **g**enerator (ICG). In §4, we present the **S**calable **P**arallel **R**andom **N**umber **G**enerators (`SPRNG`) library, a comprehensive tool for parallel and distributed pseudorandom number generation developed by the author. Finally in §5 we discuss open problems, and provide concluding remarks.

2. Quasirandom Number Generation

Monte Carlo methods are based on mathematical processes that utilize random numbers. The computational requirement for random numbers in Monte Carlo applications has been satisfied with two types of computational random numbers: pseudorandom numbers and quasirandom numbers. Pseudorandom numbers mimic the behavior of "real" random numbers in theoretical and empirical tests, whereas quasirandom numbers provide very uniformly distributed sets of numbers that may, in fact, perform poorly on tests of randomness. However, quasirandom numbers are more effective that pseudorandom numbers in situations where the uniform distribution of points is important. Such applications include *the* canonical Monte Carlo application, the numerical evaluation of integrals. It is the case that many non-quadrature Monte Carlo computations can mathematically be viewed as numerical quadrature, and so many other types of Monte Carlo applications have seen performance improvement when pseudorandom numbers have been replaced with quasirandom numbers. In

fact, many application areas that do not at face value seem to be anything like quadrature have been favorably impacted by he use of quasirandom numbers. These include simulations with random walkers in applications areas as diverse as heat conduction, [42], rarified gas dynamics, [6], particle transport, [56], and financial instrument evaluation, [7]. In addition, quasirandom numbers promise to improve the convergence of applications in quantum mechanics, materials science, biochemistry, and environmental remediation.

The mathematical motivation for quasirandom numbers can be found in the classic Monte Carlo application of numerical integration. We detail this for the trivial example of one-dimensional integration for illustrative simplicity. Let us assume that we are interested in the numerical value of $I = \int_0^1 f(x)\,dx$, and we seek to optimize approximations of the form $I \approx \frac{1}{N}\sum_{n=1}^{N} f(x_n)$. A solution to the optimization of the integration nodes, $\{x_n\}_{n=1}^{N}$, comes from the famous Koksma-Hlwaka inequality. Let us define the star discrepancy of a one-dimensional point set, $\{x_n\}_{n=1}^{N}$, by

$$D_N^* = D_N^*(x_1, \ldots, x_N) = \sup_{0 \le u \le 1} \left| \frac{1}{N} \sum_{n=1}^{N} \chi_{[0,u)}(x_n) - u \right| \tag{1}$$

where $\chi_{[0,u)}$ is the characteristic function of the half open interval $[0, u)$. The term $\sum_{n=1}^{N} \chi_{[0,u)}(x_n)$ counts the number of x_n's in the interval $[0, u)$, and thus $\left|\frac{1}{N}\sum_{n=1}^{N} \chi_{[0,u)}(x_n) - u\right|$ measures the difference between the actual distribution of points in the interval $[0, u)$ and the uniform distribution on $[0, u)$. By taking the supremum we are characterizing the distribution of the $\{x_n\}_{n=1}^{N}$ through it's maximal deviation from uniformity. We then have the remarkable theorem:

Theorem (Koksma-Hlwaka, [22]): if $f(x)$ has bounded variation, $V(f)$, on $[0, 1]$, and $x_1, \ldots, x_N \in [0, 1]$ have star discrepancy D_N^*, then:

$$\left| \frac{1}{N} \sum_{n=1}^{N} f(x_n) - \int_0^1 f(x)\,dx \right| \le V(f) D_N^*, \tag{2}$$

This simple bound on the integration error is a product of $V(f)$, the total variation of the integrand in the sense of Hardy and Krause, and D_N^*, the star discrepancy of the integration points. A major area of research in Monte Carlo is variance reduction, which indirectly deals with minimizing $V(f)$. Quasirandom number generation deals with minimization of the other term.

Mathematically, quasirandom numbers produce point sets and sequences that have low discrepancy. Discrepancy is a quantitative measure of the uniformity of a point set. The star discrepancy, introduced above, is merely

one of many discrepancies which are used to measure uniformity of discrete measures, [47]. For example, the star discrepancy of a point set of N "real" random numbers in one dimension is $O(N^{-1/2}(\log\log N)^{1/2})$, while the discrepancy of N quasirandom numbers can be as low as N^{-1}.[2] In $s > 3$ dimensions it is rigorously known that the discrepancy of a point set with N elements can be no smaller than a constant depending only on s times $N^{-1}(\log N)^{(s-1)/2}$. This remarkable result of Roth, [52], has motivated mathematicians to seek point sets and sequences with discrepancies as close to this lower bound as possible.

Perhaps the best way to illustrate the difference between quasirandom numbers and pseudorandom numbers is with a picture. Thus in *Figure 1* we plot 4096 tuples produced by successive elements from the 64-bit Percus-Kalos LCG from `SPRNG` (see §3.1.2 for details). These tuples are distributed in a manner consistent with real random tuples. In *Figure 2* we see 4096 quasirandom tuples formed by taking the 2nd and 3rd dimensions from the Soboĺ sequence, a well known quasirandom sequence. It is clear that the two figures look very different and that *Figure 2* is much more uniformly distributed. Both plots have the same number of points, and the largest "hole' in *Figure 1* is much larger than in *Figure 2*. This illustrates quite effectively the qualitative meaning of low discrepancy.

The first quasirandom number sequence was proposed by Halton, [20], and is based on the Van der Corput sequence with different prime bases for each dimension. The jth element of Van der Corput sequence base b is defined as $\phi_b(j-1)$ where $\phi_b(\cdot)$ is the radical inverse function and is computed by writing $j-1$ as an integer in base b, and then flipping the digits about the ordinal (decimal) point. Thus if $j-1 = a_n \ldots a_0$ in base b, then $\phi_b(j-1) = 0.a_0 \ldots a_n$. Since this initial work, Faure, Niederreiter and Soboĺ, [16], [44], [55], chose alternate methods based on another sort of finite field arithmetic that utilizes primitive polynomials with coefficients in some prime Galois field. All of these constructions of quasirandom sequences have discrepancies that are $O(N^{-1}(\log N)^s)$, [44]. What distinguishes them is the asymptotic constant in the discrepancy, and the computational requirements for implementation. However, practice has shown that the provable size of the asymptotic constant in the discrepancy is a poor predictor of the actual computational discrepancy displayed by a concrete implementation of any of these quasirandom number generators. There are existing implementations of the Halton, Faure, Niederreiter and Soboĺ sequences, [1], [2], [17], that are computationally efficient. Each of these sequences is

[2]Of course, the N optimal quasirandom points in $[0,1)$ are the obvious: $\frac{1}{(N+1)}, \frac{2}{(N+1)}, \ldots \frac{N}{(N+1)}$.

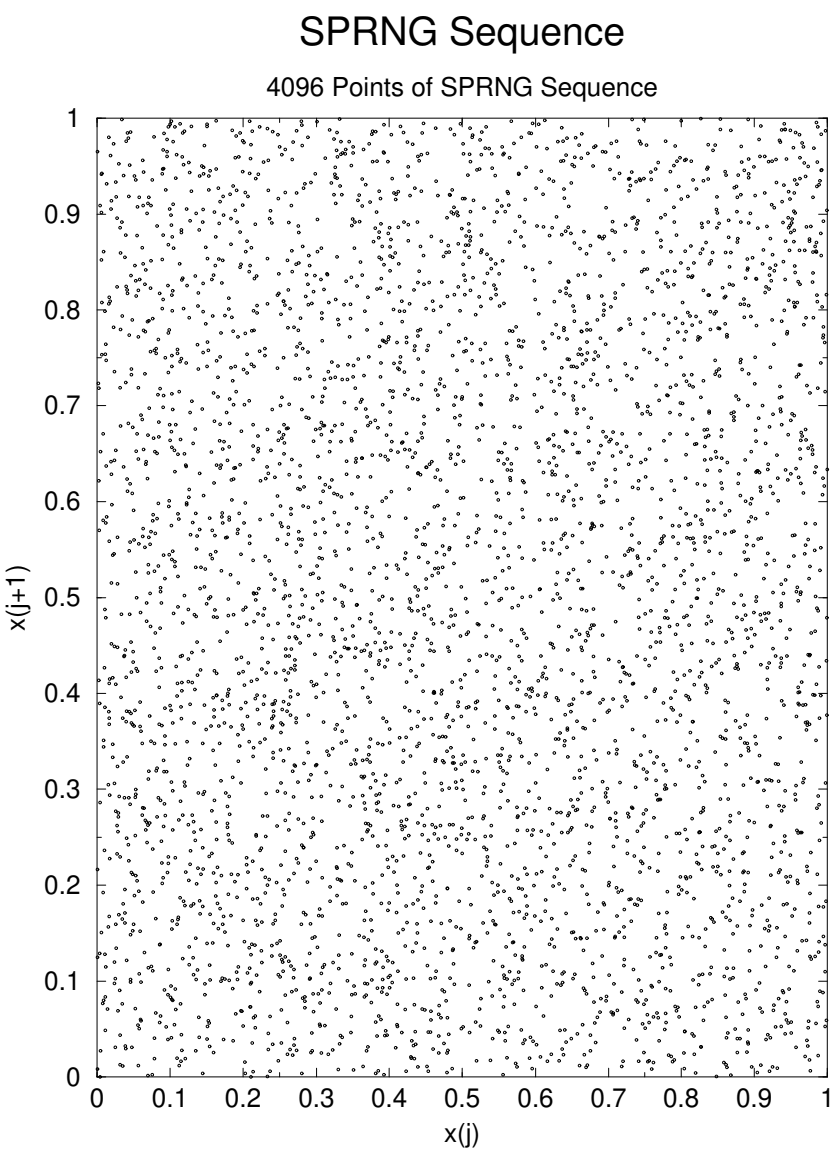

Figure 1. Tuples produced by successive elements from a `SPRNG` pseudorandom number generator.

initialized to produce quasirandom s-tuples and each one of these requires the initialization of s one-dimensional quasirandom streams.

An extremely interesting alternative to the traditional methods for quasirandom number generation is based on a well-known defect of the most commonly used pseudorandom number generator: the LCG. Recall

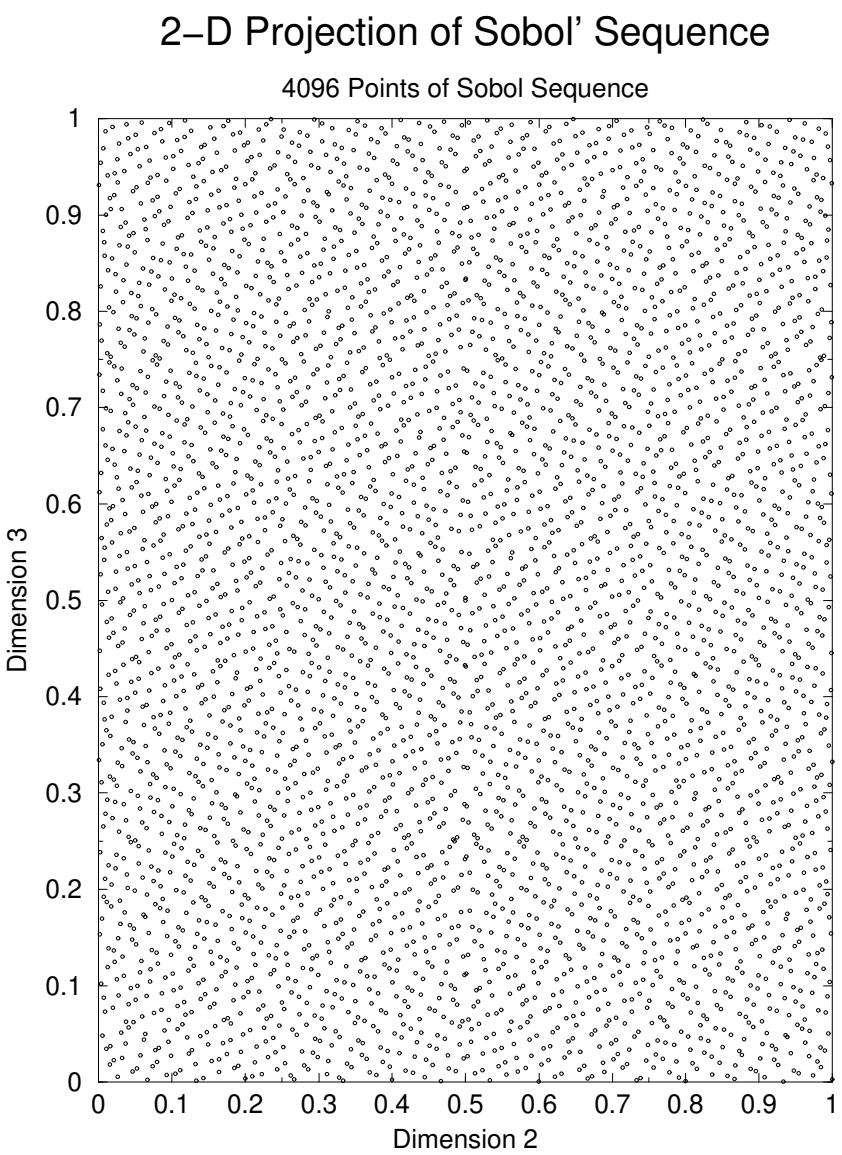

Figure 2. Tuples produced by the 2nd and 3rd dimension of the Sobol' sequence.

that an LCG has the form:

$$x_n = ax_{n-1} + b \pmod{m}. \tag{3}$$

When the multiplier, a, additive constant, b, and modulus, m, are chosen appropriately one obtains a purely periodic sequence with period as long

as $Per(x_n) = 2^k$, when m is a power-of-two, and $Per(x_n) = m - 1$, when m is prime. It is well known that s-tuples made up from LCGs lie on lattices composed of a family of parallel hyperplanes, [32]. The most widely used measure of quality for an LCG is it's performance on the spectral test, [21]. The spectral test measures the maximum spacing between hyperplanes that cover the lattice produced by overlapping s-tuples from the LCG. The maximal hyperplane spacing that is produced by the spectral test, $\frac{1}{\nu_s(a,m)}$, is a geometric constant uniquely defined by the multiplier and modulus of an LCG and the dimension of the space. The spectral test is independent of the additive constant, and so it is commonly used to measure the quality of a given multiplier-modulus pair for quadrature-like applications in s dimensions. The s-dimensional discrepancy of a point set also measures that point set's suitability for use as integration nodes. Thus it should be no surprise that LCGs with good lattice properties should produce low-discrepancy point sets. More precisely, the method of "good lattice points" (GLP) attempts to create low-discrepancy point sets by constructing s-dimensional lattices, [54]. It has been shown that the GLP figure of merit is related to the spectral test, [15], and so LCGs with good spectral test results *are* GLP integration rules when considered over their full periods.

This relationship instantly leads to a way to produce low-discrepancy point sets based on full-period LCG tuples. Based on theoretical and implementational reasons, the moduli for LCGs are chosen either to be powers-of-two or primes. By choosing a prime-modulus, m, for an LCG to be used in this form of quasirandom number generation, one is implicitly choosing a point set of $m - 1$ s-tuples. This is clearly very restrictive. On the other hand, if one picks $m = 2^k$ as the LCG modulus, then considerably more flexibility is achieved in the size of the point sets available. Let us assume that a and b are chosen so that the $\{x_n\}$ produced by the LCG have the maximal period: $Per(x_n) = 2^k$. The conditions for this are that $a \equiv 1 \pmod 4$ and $b \equiv 1 \pmod 2$. Now consider the j least-significant bits of the $\{x_n\}$ by defining $y_n = x_n \pmod{2^j}$ with $1 \leq j \leq k$. It is easy to show that the $\{y_n\}$ can be produced directly by an LCG with $a = a \pmod{2^j}$, $b = b \pmod{2^j}$, and $m = 2^j$ in equation (3).

Since LCGs modulo a power-of-two have $Per(x_n) = 2^j$, the use of overlapping s-tuples of LCGs leads to quasirandom point sets of size 2^j. If one knows that approximately 2^j quasirandom points will be required in a given calculation, this is fine. However, it is usually the case that at most one knows only the error required in the computation, *a priori*. In such a situation having access only to quasirandom point sets with power-of-two sizes seems quite restrictive. In fact, properties of LCGs make this not nearly as problematic. Note that the conditions on a and b required

to achieve the maximal period are properties only of the least significant bits of a and b. Thus one can choose a and b such that $a = a \pmod{2^j}$, $b = b \pmod{2^j}$ lead to $Per(x_n) = 2^j$ for all $j \geq 2$! Thus one can hope to find a single multiplier-additive constant pair that will provide good quality quasirandom point sets for all reasonable powers of two.

3. Parallel Pseudorandom Number Generation

In this next, rather extensive, section we will look at several methods for parallel pseudorandom number generation. Most of the methods we will present will be based on some kind of parameterization of the generators.

3.1. LINEAR CONGRUENTIAL GENERATORS

The most commonly used generator for pseudorandom numbers is the LCG. The LCG was first proposed for use by Lehmer, [27], and is referred to as the Lehmer generator in the early literature. The linear recursion underlying LCGs is given above in 3, where the constants in the recursion are likewise defined. The x_n's in 3 are integer residues modulo m, and a uniform pseudorandom number in [0,1] is produced via $z_n = x_n/m$, and he initial value of the LCG, x_0, is often called the seed.

The most important parameter of an LCG is the modulus, m. Its size constrains the period, and for implementational reasons it is always chosen to be either prime or a power-of-two. Based on which type of modulus is chosen, there is a different parameterization method. When m is prime, a method based on using the multiplier, a, as the parameter has been proposed. The rationale for this choice is outlined in, [37], and leads to several interesting computational problems.

3.1.1. *Prime Modulus*

Given we wish to parameterize a when m is prime we must determine first the family of permissible a's. A condition on a when m is prime to obtain the maximal period (of length $m-1$ in this case) is that a must be a primitive element modulo m, [21].[3] Given primitivity, one can use the following fact: if a and α are primitive elements modulo m then $\alpha = a^i \pmod{m}$ for some i relatively prime to $\phi(m)$. Note that when m is prime that $\phi(m) = m-1$. Thus a single, reference, primitive element, a, and an explicit enumeration of the integers relatively prime to $m-1$ furnish an explicit parameterization for the jth primitive element, a_j as $a_j = a^{\ell_j} \pmod{m}$ where ℓ_j is the jth integer relatively prime to $m-1$. Given an

[3] An integer, a, is primitive modulo m if the set of integers $\{a^i \pmod{m} | 1 \leq i \leq m-1\}$ equals the set $\{1 \leq i \leq m-1\}$.

explicit factorization of $m-1$, [5], efficient algorithms for computing ℓ_j can be found in a recent work of the author, [37]. An interesting open question in this regard is whether the overall efficiency of this PPRNG is minimized by choosing the prime modulus to minimize the cost of computing ℓ_j or to minimize the cost of modular multiplication modulo m.

Given this scheme there are some positive and negative features to be mentioned. A motivation for this scheme is that a common theoretical measure of the correlation among parallel streams predicts little correlation. This measure is based on exponential sums. Exponential sums are of interest in many areas of number theory. We define the exponential sum for the sequence of residues modulo m, $\{x_n\}_{n=0}^{k-1}$, as:

$$C(k) = \sum_{n=0}^{k-1} e^{\frac{2\pi i}{m} x_n}. \tag{4}$$

If the x_n are periodic and k is the period, then 4 is called a full-period exponential sum. If x_n is periodic and k is less than the full period, then 4 is a partial-period exponential sum. Examining 4 shows it to be a sum of k quantities on the unit circle. A trivial upper bound is thus $|C(k)| \leq k$. If the sequence $\{x_n\}$ is indeed uniformly distributed, then we would expect $|C(k)| = O(\sqrt{k})$, [23]. Thus the desire is to show that exponential sums of interest are neither too big nor too small to reassure us that the sequence in question is theoretically equidistributed.

Since we are interested in studying sequences for use in parallel, we must consider the cross-correlations among the sequences to be used on different processors. If $\{x_n\}$ and $\{y_n\}$ are two sequences of interest then their exponential sum cross-correlation is given by:

$$C(i,j,k) = \sum_{n=0}^{k-1} e^{\frac{2\pi i}{m}(x_{i+n} - y_{j+n})}. \tag{5}$$

Here the sum has k terms and begins with x_i and y_j.

In a previous work we only considered full-period exponential sum cross-correlation for studying these issues for a different recursion, [51]. We will take the same approach here. Suppose we have j full-period LCGs defined by $x_{k_n} = a^{\ell_i} x_{k_{n-1}} \pmod{m}$, $0 \leq k < j$. All of the pairwise full-period exponential sum cross-correlations are known to satisfy, [53]:

$$|C(m)| \leq \left(\left[\max_k \ell_k \right] - 1 \right) \sqrt{m}. \tag{6}$$

The choice of the exponents, ℓ_k, that minimizes 6 is to make ℓ_j the jth integer relatively prime to $m-1$. This necessitates an algorithm to compute

this jth integer relatively prime to an integer with known factorization, $m-1$. This is discussed at great length in [37]; however, two important open questions remain: (1) is it more efficient overall to choose m to be amenable to fast modular multiplication or fast calculation of the jth integer relatively prime to $m-1$, and (2) does the good interstream correlation of 6 also ensure good intrastream independence via the spectral test? The first of these questions is of practical interest to performance, the second; however, if answered negatively, makes such techniques less attractive for parallel pseudorandom number generation.

3.1.2. *Power-of-two Modulus*

An alternative way to use LCGs to make a PPRNG is to parameterize the additive constant in equation 3 when the modulus is a power-of-two, i.e., to $m = 2^k$ for some integer $k > 1$. This is a technique first proposed by Percus and Kalos, [50], to provide a PPRNG for the NYU Ultracomputer. It has some interesting advantages over parameterizing the multiplier; however, there are some considerable disadvantages in using power-of-two modulus LCGs.

The parameterization chooses a set of additive constants $\{b_j\}$ that are pairwise relatively prime, i.e. $\gcd(b_i, b_j) = 1$ when $i \neq j$. A prudent choice is to let b_j be the jth prime. This both ensures the pairwise relative primality and is the largest set of such residues. With this choice certain favorable interstream properties can be theoretically derived from the spectral test, [50]. However, this choice necessitates a method for the difficult problem of computing the jth prime. In their paper, Percus and Kalos do not discuss this aspect of their generator in detail, partly due to the fact that they expect to provide only a small number of PRNGs. When a large number of PPRNGs are to be provided with this method, one can use fast algorithms for the computation of $\pi(x)$, the number of primes less than x, [10], [24] . This is the inverse of the function which is desired, so we designate $\pi^{-1}(j)$ as the jth prime. The details of such an implementation need to be specified, but a very related computation for computing the jth integer relatively prime to a given set of integers is given in, [37]. It is believed that the issues for computing $\pi^{-1}(j)$ are similar.

One important advantage of this parameterization is that there is an interstream correlation measure based on the spectral test that suggests that there will be good interstream independence. Given that the spectral test for LCGs essentially measures the quality of the multiplier, this sort of result is to be expected. A disadvantage of this parameterization is that to provide a large number of streams, computing $\pi^{-1}(j)$ will be necessary. Regardless of the efficiency of implementation, this is known to be a difficult

computation with regards to its computational complexity. Finally, one of the biggest disadvantages to using a power-of-two modulus is the fact the least significant bits of the integers produced by these LCGs have extremely short periods. If $\{x_n\}$ are the residues of the LCG modulo 2^k, with properly chosen parameters, $\{x_n\}$ will have period 2^k. However, $\{x_n \pmod{2^j}\}$ will have period 2^j for all integers $0 < j < k$, [21]. In particular, this means the least-significant bit of the LCG with alternate between 0 and 1. This is such a major short coming, that it motivated us to consider parameterizations of prime modulus LCGs as discussed in §3.1.1.

3.2. LINEAR MATRIX GENERATORS

Recent trends in computer architecture have motivated researchers to study methods of generating pseudorandom vectors, [45], [46]. These techniques are appropriate to vector architectures, but are not well suited to parallel machines due to the algorithm's lack of data locality. **L**inear **m**atrix **g**enerator**s** (LMGs) are given by the following equation:

$$\mathbf{x}_n = \mathbf{A}\mathbf{x}_{n-1} \pmod{m}. \tag{7}$$

Here the matrix $\mathbf{A}$ is $k \times k$ and the vector $\mathbf{x}_n$ is k-dimensional. One obtains a uniform pseudorandom vector by forming $\mathbf{z}_n = \mathbf{x}_n/m$. When m is prime, the maximal period for the LMG is $m^k - 1$ if and only if $\mathbf{A}$ has a characteristic polynomial that is primitive modulo m, [47]. In this situation the k-tuples produced by equation 7 will pass the k-dimensional equidistribution test as well as an LCG passed the one-dimensional equidistribution test. This is to be expected as the full-period of the LMG produces all possible k-tuples modulo m. For dimensions larger than k, behavior analogous to the LCG in two or more dimensions is seen.

3.3. SHIFT-REGISTER GENERATORS

Shift **r**egister **g**enerator**s** (SRGs) are linear recursions modulo 2, [19], [28], [57], of the form:

$$x_{n+k} = \sum_{i=0}^{k-1} a_i x_{n+i} \pmod{2}, \tag{8}$$

where the a_i's are either 0 or 1. An alternative way to describe this recursion is to specify the kth degree binary characteristic polynomial, [29]:

$$f(x) = x^k + \sum_{i=0}^{k-1} a_i x^i \pmod{2}. \tag{9}$$

To obtain the maximal period of $2^k - 1$, a sufficient condition is that $f(x)$ be a primitive kth degree polynomial modulo 2. If only a few of the a_i's are 1, then 8 is very cheap to evaluate. Thus people often use known primitive trinomials to specify SRG recursions. This leads to very efficient, two-term, recursions.

There are two ways to make pseudorandom integers out of the bits produced by 8. The first, called the digital multi-step method, takes successive bits from 8 to form an integer of desired length. Thus, with the digital multi-step method, it requires n iterations of 8 to produce a new n-bit pseudorandom integer. The second method, called the generalized feedback shift-register, creates a new n-bit pseudorandom integer for every iteration of 8. This is done by constructing the n-bit word from x_{n+k} and $n-1$ other bits from the k bits of SRG state. While these two methods seem different, they are very related, and theoretical results for one always hold for the other. Reader's interested in more general information on SRGs should consult the references: [19], [28], [57]. One way to parameterize SRGs is analogous to the LCG parameterization discussed in §3.1.1. There we took the object that made the LCG full-period, the primitive root multiplier, and found a representation for all of them. Using this analogy we identify the primitive polynomial in the SRG as the object to parameterize. We begin with a known primitive polynomial of degree k, $p(x)$. It is known that only certain decimations of the output of a maximal-period shift register are themselves maximal and unique with respect to cyclic reordering, [29]. We seek to identify those. The number of decimations that are both maximal-period and unique when $p(x)$ is primitive modulo 2 and k is a Mersenne exponent is $\frac{2^k-2}{k}$. If a is a primitive root modulo the prime $2^k - 1$, then the residues $a^i \pmod{2^k-1}$ for $i = 1$ to $\frac{2^k-2}{k}$ form a set of all the unique, maximal-period decimations. Thus we have a parameterization of the maximal-period sequences of length $2^k - 1$ arising from primitive degree k binary polynomials through decimations.

The entire parameterization goes as follows. Assume the kth stream is required, compute $d_k \equiv a^k \pmod{2^k-1}$ and take the d_kth decimation of the reference sequence produced by the reference primitive polynomial, $p(x)$. This can be done quickly with polynomial algebra. Given a decimation of length $2k+1$, this can be used as input the Berlekamp-Massey algorithm to recover the primitive polynomial corresponding to this decimation. The Berlekamp-Massey algorithm finds the minimal polynomial that generates a given sequence, [41] in time linear in k.

This parameterization is relatively efficient when the binary polynomial algebra is implemented correctly. However, there is one major drawback to using such a parameterization. While the reference primitive polyno-

mial, $p(x)$, may be sparse, the new polynomials need not be. By a sparse polynomial we mean that most of the a_i's in 8 are zero. The cost of stepping 8 once is proportional to the number of non-zero a_i's in 8. Thus we can significantly increase the bit-operational complexity of a SRG in this manner.

The fact that the parameterization methods for prime modulus LCGs and SRGs are so similar is no accident. Both are based on maximal period linear recursions over a finite field. Thus the discrepancy and exponential sum results for both the types of generators are similar, [47]. However, a result for SRGs analogous to that in 6 is not known. It is open whether or not such a cross-correlation result holds for SRGs, but it is widely thought to.

3.4. LAGGED-FIBONACCI GENERATORS

In the previous sections we have discussed generators that can be parallelized by varying a parameter in the underlying recursion. In this section we discuss the **a**dditive **l**agged-**F**ibonacci **g**enerator (ALFG) : a generator that can be parameterized through its initial values. The ALFG can be written as:

$$x_n = x_{n-j} + x_{n-k} \pmod{2^m}, \quad j < k. \tag{10}$$

In recent years the ALFG has become a popular generator for serial as well as scalable parallel machines, [31]. In fact, the generator with $j = 5$, $k = 17$, and $m = 32$ was the standard PPRNG in Thinking Machines Connection Machine Scientific Subroutine Library. This generator has become popular for a variety of reasons: (1) it is easy to implement, (2) it is cheap to compute using 10, and (3) the ALFG does well on standard statistical tests, [34].

An important property of the ALFG is that the maximal period is $(2^k - 1)2^{m-1}$. This occurs for very specific circumstances, [4], [35], from which one can infer that this generator has $2^{(k-1)\times(m-1)}$ different full-period cycles, [39]. This means that the state space of the ALFG is toroidal, with equation 10 providing the algorithm for movement in one of the torus dimension. It is clear that finding the algorithm for movement in the other dimension is the basis of a very interesting parameterization. Since equation 10 tells us how to cycle over the full period of the ALFG, one must find a seed that is not in a given full-period cycle to move in the second dimension. The key to moving in this second dimension is to find an algorithm for computing seeds in any given full-period cycle.

A very elegant algorithm for movement in this second dimension is based on a simple enumeration as follows. One can prove that the initial seed,

$\{x_0, x_1, \ldots, x_{k-1}\}$, can be bit-wise initialized using the following template:

m.s.b.				l.s.b.	
b_{m-1}	b_{m-2}	...	b_1	b_0	
■	■	...	0	0	x_{k-1}
0	■	...	■	0	x_{k-2}
⋮	⋮	⋮	⋮	⋮	
■	0	...	■	0	x_1
■	■	...	■	1	x_0

(11)

Here each square is a bit location to be assigned. Each unique assignment gives a seed in a provably distinct full-period cycle, [39]. Note that here the least-significant bits, b_0 are specified to be a fixed, non-zero, pattern. If one allows an $O(k^2)$ precomputation to find a particular least-significant-bit pattern then the template is particularly simple:

m.s.b.				l.s.b.	
b_{m-1}	b_{m-2}	...	b_1	b_0	
■	■	...	■	b_{0k-1}	x_{k-1}
■	■	...	■	b_{0k-2}	x_{k-2}
⋮	⋮	⋮	⋮	⋮	
■	■	...	■	b_{01}	x_1
0	0	...	0	1	x_0

(12)

Given the elegance of this explicit parameterization, one may ask about the exponential sum correlations between these parameterized sequences. It is known that certain sequences are more correlated than others as a function of the similarity in the least-significant bits in the template for parameterization, [40]. However, it is easy to avoid all but the most uncorrelated pairs in a computation, [51]. In this case there is extensive empirical evidence that the full-period exponential sum correlation between streams is $O(\sqrt{(2^k-1)2^{m-1}})$, the square root of the full-period. This is essentially optimal. Unfortunately, there is no analytic proof of this result, and improvement of the best known analytic result, [40], is an important open problem in the theory of ALFGs.

Another advantage of the ALFG is that one can implement these generators directly with floating-point numbers to avoid the constant conversion from integer to floating-point that accompanies the use of other generators. This is a distinct speed improvement when only floating-point numbers are required in the Monte Carlo computation. However, care must be taken to maintain the identity of the corresponding integer recursion when using the

floating-point ALFG in parallel to maintain the uniqueness of the parallel streams. A discussion of how to ensure fidelity with the integer streams can be found in [3].

An interesting cousin of the ALFG is the **m**ultiplicative **l**agged-**F**ibonacci **g**enerator (MLFG). It is defined by:

$$x_n = x_{n-j} \times x_{n-k} \pmod{2^m}, \quad j < k. \tag{13}$$

While this generator has a maximal-period of $(2^k - 1)2^{m-3}$, which is a quarter the length of the corresponding ALFG, [35], it has empirical properties considered to be superior to ALFGs, [34]. Of interest for parallel computing is that a parameterization analogous to that of the ALFG exists for the MLFG, [36].

3.5. INVERSIVE CONGRUENTIAL GENERATORS

An important new type of PRNG that, as yet, has not found any widely distributed implementation is the **i**nversive **c**ongruential **g**enerator (ICG). This generator comes in two versions, the recursive ICG, [14], [43], and the explicit ICG, [48] . The formula for the recursive ICG is:

$$x_n = a\overline{x}_{n-1} + b \pmod{m}, \tag{14}$$

while the explicit ICG has formula:

$$x_n = \overline{an + b} \pmod{m}. \tag{15}$$

In both the above equations $\bar{c}$ denotes the multiplicative inverse modulo m in the sense that $c\bar{c} \equiv 1 \pmod{m}$ when $c \neq 0$, and $\bar{0} = 0$.

An advantage of ICGs over LCGs are that tuples made from ICGs do not fall in hyperplanes, [32], [33]. The quantification of this is the lattice test. We say that a generator passes the k-dimensional lattice test if vectors made up of k-tuples from the generator span the k-dimensional vector space taken modulo m. ICGs have optimal behavior for the lattice test in that they pass for $k \leq m - 1$. Another advantage of ICGs is that they are not nearly as uniformly distributed over their full period as generators from linear recursions. They behave more like truly random numbers, [47].

An interesting fact about the (non)lattice structure of tuples from explicit ICGs has ramifications for PPRNGs via parameterization. Consider the parameterized explicit ICG: $x_{k_n} = \overline{a_k n + b_k} \pmod{m}$, $0 \leq k \leq s$. If the residues modulo m, $a_0\overline{b_0}, \ldots, a_{s-1}\overline{b_{s-1}}$, are all distinct, then set of all the s-tuples $(x_{0_j}, x_{1_j}, \ldots, x_{s-1_j})$, $j = 0$ to $m-1$, appear to be extremely well distributed as follows. Take any s-dimensional hyperplane passing through the origin, it will intersect at most $s - 1$ of these points.

While ICGs have some very compelling equidistribution properties, they remain out of mainstream random number packages. This is due to both the fact that ICGs are relatively unknown outside of the mathematical random number generation community and that the cost of doing modular inversion is quite considerable. If we consider the cost of modular multiplication to be the cost unit, then modular inversion costs $O(\log_2 m)$ units. For certain applications, this extra cost may be worth it, but in most applications that involve parallel computers computational efficiency is an important factor. Thus the author expects to see ICGs available in some serial random number packages; however, he doubts if ICGs will be implemented for parallel machines any time soon.

4. SPRNG

The SPRNG library has been developed with the support of DARPA under the Scalable Libraries Initiative run by the Information Technology Office (ITO).[4] SPRNG is currently in it's first, full, Version 1.0 release. Moreover SPRNG is now supported and maintained by NCSA under their high-performance software activities funded by the NSF under PACI. In addition, there has been considerable interest from most of the high-performance computing vendors in using SPRNG as a common, parallel pseudorandom number generation library on their machines. Thus SPRNG, itself, will be a lasting contribution to mathematical software for parallel Monte Carlo computations.

SPRNG is designed to use parameterized pseudorandom number generators to provide random number streams to parallel processes. SPRNG includes the following:

- Several, qualitatively distinct, well tested, scalable RNGs
- Initialization without interprocessor communication
- Reproducibility by using the parameters to index the streams
- Reproducibility controlled by a single "global" seed
- Minimization of interprocessor correlation with the included generators
- A uniform C, C++, FORTRAN, and MPI interface
- Extensibility
- An integrated test suite including physical tests

The decision to use parameterized generators was based on work of the author in parameterizing several different, common, RNGs to provide full-period streams of random numbers for each, unique, parameter value.

[4]DARPA Contract Number DABT63-95-C-0123 for ITO: Scalable Systems and Software, entitled *A Scalable Pseudorandom Number Generation Library for Parallel Monte Carlo Computations.*

These generators then formed the core of the generators currently available in SPRNG :

- Additive lagged-Fibonacci: $x_n = x_{n-r} + x_{n-s} \pmod{2^m}$
- Multiplicative lagged-Fibonacci: $x_n = x_{n-r} \times x_{n-s} \pmod{2^m}$
- Prime modulus multiplicative congruential: $x_n = a x_{n-1} \pmod{m}$
- Power-of-two modulus linear congruential: $x_n = a x_{n-1} + b \pmod{2^m}$
- Combined multiple recursive generator: $z_n = x_n + y_n \times 2^{32}$, where x_n is a linear congruential generator modulo 2^{64} and y_n satisfies $y_n = 107374182 y_{n-1} + 104480 y_{n-5} \pmod{2147483647}$

All the above generators can be thought of as being parameterized by a simple integer valued function, $f(\cdot)$ where $f(i)$ gives the appropriate parameter for the ith random number stream. Given this uniformity, the random number streams are mapped onto the binary tree through the canonical enumeration via the index i. This allows us to take the parameterization and use it to produce new streams from existing streams without the need for interprocessor communication. We accomplish this by allowing a given stream access only to those streams associated with the subtree rooted at the given stream. This can be used to automatically manage static and dynamic creation of streams, and prohibits reuse of streams. To permit a calculation to be redone with different random numbers, we can apply a mixing function $p_s(\cdot)$ so that we map the streams onto the binary tree via the index $p_s(i)$ instead of just i. The function $p_s(\cdot)$ is a permutation parameterized by the global seed s. Different values of s give different permutations and thus map the streams onto the binary tree in different yet distinct ways.

SPRNG was also designed to be flexible, and to be as easy to use as possible. The Monte Carlo community is very conservative, and many groups use RNGs that have been handed down the generations (sometimes all the way back to Lehmer or Metropolis!). Thus we not only developed the library in collaboration with a member of this conservative community, but we added the ability to extend the library with a user supplied generator. Thus a user may add their own RNG by rewriting two dummy SPRNG two routines and recompiling SPRNG. This then gives a user access to their own generator within the SPRNG parallel infrastructure. This is a powerful capability, and our own implementational experience has shown that any implementation must be thoroughly tested, empirically, to prevent unforeseen correlations within streams. (We found such unanticipated correlations ourselves in very carefully thought out implementations). Thus SPRNG includes a comprehensive testing suite to validate new generators. Together, the extensibility and testing suite aids both users wanting to implement their own generators in parallel, and provides library developers a powerful

rapid prototyping tool.

Through the default generators, SPRNG is a tool for parallel pseudorandom number generation. The results obtained are also reproducible, and SPRNG provides a simple way to run on distributed-memory parallel machines using popular languages and parallel paradigms and supports distribution on a heterogeneous collection of machines.[5] When a different RNG is desired, e.g. when a particular RNG is thought to give spurious results in a given application, a qualitatively different generator can replace the original by merely relinking the user program with SPRNG. Finally, new RNGs can be incorporated into SPRNG with little more than coding the generation and initialization routines and recompiling SPRNG.

5. Conclusions and Open Problems

We have presented a considerable amount of detail about random number generation. We have introduced the reader to quasirandom (low discrepancy) numbers, and have discussed pseudorandom number generation for parallel and distributed systems. In particular, we have described the SPRNG library as an example of a comprehensive library for parallel Monte Carlo.

While care has been taken in constructing generators for the SPRNG package, the designers realize that there is no such thing as a PRNG that behaves flawlessly for every application. This is even more true when one considers using scalable platforms for Monte Carlo. The underlying recursions that are used are for PRNGs are simple, and so they inevitably have regular structure. This deterministic regularity permits analysis of the sequences and is the PRNG's Achilles heel. Thus any large Monte Carlo calculation must be viewed with suspicion as an unfortunate interplay between the application and PRNG may result in spurious results. The only way to prevent this is to treat each new Monte Carlo derived result as an experiment that must be controlled. The tools required to control problems with the PRNG include the ability to use another PRNG in the same calculation. In addition, one must be able to develop and use entirely new PRNGs as well. These capabilities as well as parallel and serial tests of randomness, [8], are components that make the SPRNG package unique among tools for parallel Monte Carlo.

References

1. P. Bratley and B. L. Fox, "Algorithm 659; Implementing Sobol's quasirandom

[5]In fact, the developers of CONDOR, a distributed computing tool, plan to incorporate SPRNG directly into CONDOR to make CONDOR a comprehensive tool for Monte Carlo on distributed heterogeneous collections of machines, [30].

sequence generator," *ACM Trans. on Math. Software*, **14**: 88–100, 1988.
2. P. Bratley, B. L. Fox and H. Niederreiter, "Implementation and tests of low-discrepancy point sets," *ACM Trans. on Modeling and Comp. Simul.*, **2**: 195–213, 1992
3. R. P. Brent, "Uniform Random Number Generators for Supercomputers" in *Proceedings Fifth Australian Supercomputer Conference*, 5th ASC Organizing Committee, pp. 95–104, 1992.
4. R. P. Brent, "On the periods of generalized Fibonacci recurrences," *Mathematics of Computation*, **63**: 389–401, 1994.
5. J. Brillhart, D. H. Lehmer, J. L. Selfridge, B. Tuckerman and S. S. Wagstaff, Jr., "Factorizations of $b^n \pm 1$ $b = 2, 3, 5, 7, 10, 11, 12$ up to high powers," *Contemporary Mathematics* Volume 22, Second Edition, American Mathematical Society, Providence, Rhode Island, 1988.
6. R. E. Caflisch, "Monte Carlo and quasi-Monte Carlo methods," *Acta Numerica*, **7**: 1–49, 1998.
7. R. E. Caflisch, W. Morokoff, A. B. Owen, "Valuation of mortgage-backed securities using Brownian bridges to reduce effective dimension," *J. Comput. Finance*, **1**: 27–46, 1997.
8. S. A. Cuccaro, M. Mascagni and D. V. Pryor, "Techniques for testing the quality of parallel pseudorandom number generators," in *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, Philadelphia, Pennsylvania, pp. 279–284, 1995.
9. I. Deák, "Uniform random number generators for parallel computers," *Parallel Computing*, **15**: 155–164, 1990.
10. M. Deleglise and J. Rivat, "Computing $\pi(x)$: the Meissel, Lehmer, Lagarias, Miller, Odlyzko method," *Mathematics of Computation*, **65**: 235–245, 1996.
11. A. De Matteis and S. Pagnutti, "Parallelization of random number generators and long-range correlations," *Parallel Computing*, **15**: 155–164, 1990.
12. A. De Matteis and S. Pagnutti, "A class of parallel random number generators," *Parallel Computing*, **13**: 193–198, 1990.
13. A. De Matteis and S. Pagnutti, "Long-range correlations in linear and non-linear random number generators," *Parallel Computing*, **14**: 207–210, 1990.
14. J. Eichenauer and J. Lehn, "A nonlinear congruential pseudorandom number generator," *Statist. Hefte*, **37**: 315–326, 1986.
15. K. Entacher, P. Hellekalek and P. L'Ecuyer, "Quasi-Monte Carlo integration with linear congruential generators," presented at *Third International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, and personal communication, 1998.
16. H. Faure, "Discrépance de suites associées à un système de numération (en dimension s)," *Acta Arithmetica*, **XLI**: 337–351, 1992.
17. B. L. Fox, "Algorithm 647; Implementation and relative efficiency of quasirandom sequence generators," *ACM Trans. on Math. Software*, **12**: 362–376, 1986.
18. P. Frederickson, R. Hiromoto, T. L. Jordan, B. Smith and T. Warnock, "Pseudo-random trees in Monte Carlo," *Parallel Computing*, **1**: 175–180, 1984.
19. S. W. Golomb, *Shift Register Sequences*, Revised Edition, Aegean Park Press, Laguna Hills, California, 1982.
20. J. H. Halton, "On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals," *Numer. Math.*, **2**: 84–90, 1960.
21. D. E. Knuth, *The Art of Computer Programming: Volume 2, Seminumerical Algorithms*, Third Edition, Addison-Wesley: Reading, MA, 1998.
22. J. F. Koksma, "Een algemeene stelling uit de theorie der gelijkmatige verdeeling modulo 1," *Mathematica B (Zutphen)*, **11**: 7–11, 1942/43.
23. L. Kuipers and H. Niederreiter, *Uniform distribution of sequences*, John Wiley and Sons: New York, 1974.

24. J. C. Lagarias, V. S. Miller and A. M. Odlyzko, "Computing $\pi(x)$: The Meissel-Lehmer method," *Mathematics of Computation*, **55**: 537–560, 1985.
25. P. L'Ecuyer, "Random numbers for simulation," *Communications of the ACM*, **33**: 85–97, 1990.
26. P. L'Ecuyer and S. Côté, "Implementing a random number package with splitting facilities," *ACM Trans. on Mathematical Software*, **17**: 98–111, 1991.
27. D. H. Lehmer, "Mathematical methods in large-scale computing units," in *Proc. 2nd Symposium on LargeScale Digital Calculating Machinery*, Harvard University Press: Cambridge, Massachusetts, pp. 141–146, 1949.
28. T. G. Lewis and W. H. Payne, "Generalized feedback shift register pseudorandom number algorithms," *Journal of the ACM*, **20**: 456–468, 1973.
29. R. Lidl and H. Niederreiter, *Introduction to finite fields and their applications*, Cambridge University Press: Cambridge, London, New York, 1986.
30. M. Litzkow, M. Livny, and M. W. Mutka, "Condor - A Hunter of Idle Workstations," *Proceedings of the 8th International Conference of Distributed Computing Systems*, pp. 104–111, June, 1988.
31. J. Makino, "Lagged-Fibonacci random number generator on parallel computers," *Parallel Computing*, **20**: 1357–1367, 1994.
32. G. Marsaglia, "Random numbers fall mainly in the planes," *Proc. Nat. Acad. Sci. U.S.A.*, **62**: 25–28, 1968.
33. G. Marsaglia, "The structure of linear congruential sequences," in *Applications of Number Theory to Numerical Analysis*, S. K. Zaremba, Ed., Academic Press, New York, pp. 249–285, 1972.
34. G. Marsaglia, "A current view of random number generators," in *Computing Science and Statistics: Proceedings of the XVIth Symposium on the Interface*, pp. 3–10, 1985.
35. G. Marsaglia and L.-H. Tsay, "Matrices and the structure of random number sequences," *Linear Alg. and Applic.*, **67**: 147–156, 1985.
36. M. Mascagni, "A parallel non-linear Fibonacci pseudorandom number generator," abstract, 45th SIAM Annual Meeting, 1997.
37. M. Mascagni, "Parallel linear congruential generators with prime moduli," 1997 IMA Preprint #1470 and to appear in Parallel Computing, 1998.
38. M. Mascagni, "Some methods of parallel pseudorandom number generation," to appear in *Proceedings of the IMA Workshop on Algorithms for Parallel Processing*, R. Schreiber, M. Heath and A. Ranade editors, Springer-Verlag: New York, Berlin, 1998.
39. M. Mascagni, S. A. Cuccaro, D. V. Pryor and M. L. Robinson, "A fast, high-quality, and reproducible lagged-Fibonacci pseudorandom number generator," *Journal of Computational Physics*, **15**: 211–219, 1995.
40. M. Mascagni, M. L. Robinson, D. V. Pryor and S. A. Cuccaro, "Parallel pseudorandom number generation using additive lagged-Fibonacci recursions," *Springer Verlag Lecture Notes in Statistics*, **106**: 263–277, 1995.
41. J. L. Massey, "Shift-register synthesis and BCH decoding," *IEEE Trans. Information Theory*, **IT-15**: 122–127, 1969.
42. W. Morokoff and R. E. Caflisch, "A quasi-Monte Carlo approach to particle simulation of the heat equation," *SIAM J. Numer. Anal.*, **30**: 1558–1573, 1993.
43. H. Niederreiter, "Statistical independence of nonlinear congruential pseudorandom numbers," *Montash. Math.*, **106**: 149–159, 1988.
44. H. Niederreiter, "Low-discrepancy and low-dispersion sequences," *J. Number Theory*, **30**: 51–70, 1988.
45. H. Niederreiter, "Statistical independence properties of pseudorandom vectors produced by matrix generators," *J. Comput. and Appl. Math.*, **31**: 139–151, 1990.
46. H. Niederreiter, "Recent trends in random number and random vector generation," *Ann. Operations Research*, **31**: 323–346, 1991.

47. H. Niederreiter, *Random number generation and quasi-Monte Carlo methods*, SIAM: Philadelphia, Pennsylvania, 1992.
48. H. Niederreiter, "On a new class of pseudorandom numbers for simulation methods," *J. Comput. Appl. Math.*, **65**: 159–167, 1994.
49. S. K. Park and K. W. Miller, "Random number generators: good ones are hard to find," *Communications of the ACM*, **31**: 1192–1201, 1998.
50. O. E. Percus and M. H. Kalos, "Random number generators for MIMD parallel processors," *J. of Par. Distr. Comput.*, **6**: 477–497, 1989.
51. D. V. Pryor, S. A. Cuccaro, M. Mascagni and M. L. Robinson, "Implementation and usage of a portable and reproducible parallel pseudorandom number generator," in *Proceedings of Supercomputing '94*, IEEE, pp. 311–319, 1994.
52. K. F. Roth, "On irregularities of distribution," *Mathematika*, **1**: 73–79, 1954.
53. W. Schmidt, *Equations over Finite Fields: An Elementary Approach*, Lecture Notes in Mathematics #536, Springer-Verlag: Berlin, Heidelberg, New York, 1976.
54. I. H. Sloan and S. Joe, *Lattice Methods for Multiple Integration*, Oxford University Press: New York, 1994.
55. I. M. Sobol', "The distribution of points in a cube and approximate evaluation of integrals," *Zh. Vychisl. Mat. Mat. Fiz.*, **7**: 784–802, 1967.
56. J. Spanier, "Quasi-Monte Carlo methods for particle transport problems," in H. Niederreiter and P. Shiue, eds., *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, Springer-Verlag, New York: 121–148, 1995.
57. R. C. Tausworthe, "Random numbers generated by linear recurrence modulo two," *Mathematics of Computation*, **19**: 201–209, 1965.