Random Number Generators for Parallel Applications

ASHOK SRINIVASAN

National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, Illinois

DAVID M. CEPERLEY

National Center for Supercomputing Applications and Department of Physics, University of Illinois at Urbana-Champaign, Illinois

MICHAEL MASCAGNI

Program in Scientific Computing and Department of Mathematics, University of Southern Mississippi, Mississippi

1 Introduction

Random numbers arise in computer applications in several different contexts, such as : (i) In the Monte Carlo method to estimate a many-dimensional integral by sampling the integrand. Metropolis Monte Carlo or, more generally, Markov Chain Monte Carlo (MCMC), to which this volume is mainly devoted, is a sophisticated version of this where one uses properties of random walks to solve problems in high dimensional spaces, particularly those arising in statistical mechanics, (ii) In modeling random processes in nature such as those arising in ecology or economics. (iii) In cryptography, one uses randomness to hide information from others. (iv) Random numbers may also be used in games, for example during interaction with the user.

It is only the first class of applications to which this article is devoted, because these computations require the highest quality of random numbers. The ability to do a multidimensional integral relies on properties of uniformity of n-tuples of random numbers and/or the equivalent property that random numbers be uncorrelated. The quality aspect in the other uses is normally less important simply because the models are usually not all that precisely specified. The largest uncertainties are typically due more to approximations arising in the formulation of the model than those caused by lack of randomness in the random number generator.

In contrast, the first class of applications can require very precise solutions. Increasingly, computers are being used to solve very well-defined but hard mathematical problems. For example, as Dirac [1] observed in 1929, the physical laws necessary for the mathematical theory of a large part of physics and the whole of chemistry are completely known and it is only necessary to find precise methods for solving the equations for complex systems. In the intervening years fast computers and new computational methods have come into existence. In quantum chemistry, physical properties must be calculated to "chemical accuracy" (say 0.001 Rydbergs) to be relevant to physical properties. This often requires a relative accuracy of 10^5 or better. Monte Carlo methods are used to solve the "electronic structure problem" often to high levels of accuracy [2] (see also articles by Reynolds, Nightingale, and Kalos in this volume). In these methods one can use from 10^7 to 10^{12} random numbers, and subtle correlations between these numbers could lead to significant errors.

Another example is from the numerical study of phase transitions. Renormalization theory has proven accurate for the basic scaling properties of simple transitions. The attention of the research community is now shifting to corrections to scaling, and to more complex models. Very long simulations (also of the MCMC type) are done to investigate this effect and it has been discovered that the random number generator can influence the results [3, 4, 5, 6]. As computers become more powerful, and Monte Carlo methods become more commonly used and more central to scientific progress, the quality of the random number sequence becomes more important.

Given that the quality (which we shall define in a moment) of random numbers is becoming more and more important, the unfortunate fact is that important aspects of quality are very difficult to prove mathematically. The best one can do today is test empirically. But an empirical test is always finite. We will report here tests on random number streams that are of record length (up to about 10^{12} numbers). However, they will have to be redone in a few years with even longer sequences. Also, important algorithms use random numbers in a way that is hard to encapsulate in a test for which we know the answer, and so we must resort to general guidelines on safe ways to use random number generators in practice.

This article constitutes a brief review of recent developments in random number generation. There are several excellent reviews of the older literature. In particular we recommend the reader seriously interested in random number generation read the lengthy introduction in Knuth [7] and the shorter introduction in the 2nd edition of Numerical Recipes [8]. More information can also be found in references [9, 10, 11].

We shall focus here on developments caused by widespread use of parallel computers to perform Monte Carlo calculations. Our impression is that individual users are porting random number generators to parallel computers in an ad hoc fashion, possibly unaware of some of the issues which come to the fore when massive calculations are performed. Parallel algorithms can probe other qualities of random number generators such as inter-process correlation. There is a recent review which covers parallel random number generation in somewhat more depth by Coddington [12]. The interested reader can also refer to [13, 14, 15, 16, 17] for work related to parallel random number generation and testing.

This article is structured as follows. First we discuss the desired properties that random number generators should have. Next we discuss several methods that have been used as generators in particular on parallel computers. Then we discuss testing procedures and show some results of our extensive tests. Quasi random numbers (QRN) have recently been introduced as a way to achieve faster convergence than true random numbers. We briefly discuss these and give some guidelines concerning those applications for which they are likely to be most effective.

We have recently developed a library implementing several of the parallel random number generators and statistical tests of them on the most widely available multiprocessor computers. Documentation and software are available at:

http://www.ncsa.uiuc.edu/Apps/CMP/RNG/RNG-home.html.

2 Desired Properties of Random Number Generators

In this section we discuss some of the desired properties of good random number generators. We shall then explain specific implications of these for parallel random number generation.

First of all let us define a random number sequence, $\{u_i\}$ where i is an integer. In this article we will be concerned exclusively with uniformly distributed numbers. Other distributions can be generated by standard techniques [18]. Uniform numbers can be either reals, by convention in (0,1) such as those returned by the FORTRAN **ranf** and C **drand48** functions, or integer, by convention in $[1, 2^n)$ for some n close to the word size on the computer. We shall denote the reals by u_k and the integers by I_k. Clearly, for many purposes integer and real generators on a computer are virtually equivalent if n is large enough and if we define $u = I2^{-n}$.

A. Randomness:

Let us consider the following experiment to verify the randomness of an infinite sequence of integers in $[1, d]$. Suppose we let you view as many numbers from the sequence as you wished to. You should then guess any other number in the sequence. If the likelihood of your guess being correct is greater than $1/d$, then the sequence is not random. In practice, to estimate the winning probabilities we must play this guessing game several times.

This test has certain implications, the most important of which is the uniformity of the individual elements. If the sequence consists of integers in $[1, d]$, then the probability of getting any particular integer should be $1/d$. All good sequences are constructed to have this property. But applications in statistical mechanics as well as other real applications rely heavily on uniformity in higher dimensions, at least 4 dimensions if not thousands.

We define uniformity in higher dimensions as follows. Suppose we define n-tuples $U_i^n =$

```
x = 0
repeat loop N times
    xtrial = x + δ * (sprng() − 0.5)
    if (exp(−β * (V(xtrial) − V(x))) > sprng()) then
        x = xtrial
    endif
end loop
```

Figure 1: Algorithm for simulating movement of the particle.

$(u_{i+1}, ...u_{i+n})$ and divide the n-dimensional unit hypercube into many equal sub-volumes. A sequence is uniform if in the limit of an infinite sequence all the sub-volumes have an equal number of occurrences of random n-tuples. For a random sequence this will be true for *all* values of n and all partitions into subvolumes, though in practice we only test for small values of n.

The following simple Metropolis Monte Carlo example demonstrates how correlations between successive pairs of random numbers can give incorrect results. Suppose we sample the movement of a particle along the x axis confined in a potential well which is symmetric about the origin: $V(x) = V(-x)$. The classic Metropolis algorithm is outlined in Fig. 1. At each step in our calculations, the particle is moved to a trial position with the first random number (*sprng()*) and then that step is accepted or rejected with the second random number.

Fig. 2 shows the results of the particle density computed exactly (for a harmonic well) with a good random number sequence, and also with sequences which are deliberately chosen to be correlated or anti-correlated. In the latter case a high number is likely to be followed by a low number and a low number by a high number. The particle density is then not symmetric because movements to the right are more likely to be rejected than movements to the left, so that the final distribution is skewed. This occurs despite uniformity of the individual elements of the sequence.

Now let us imagine how this changes for N particles in three dimensions. The usual Metropolis MC algorithm for a simple classical fluid will use random numbers four at a time (three for

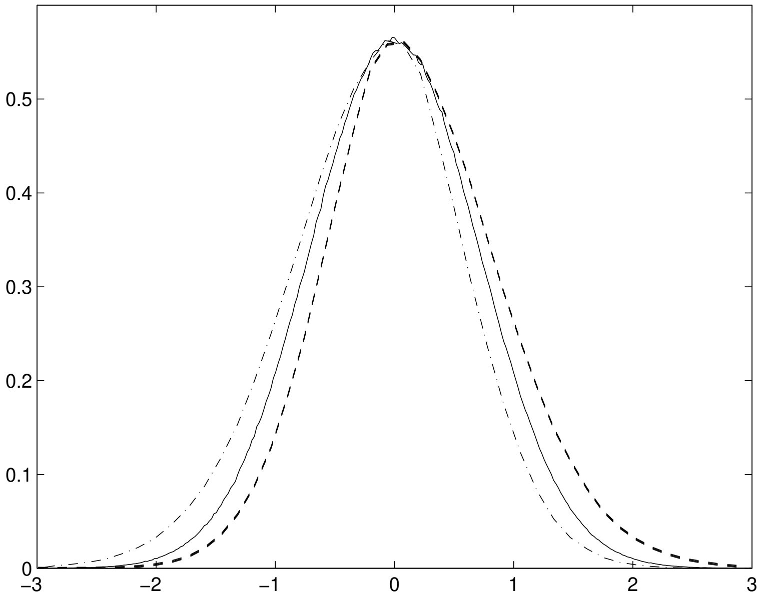

Figure 2: Distribution of particle positions in one-dimensional random walk simulations. The solid line shows the results with an uncorrelated sequence, the bold dashed line for sequences with correlation coefficient = -0.2, and the dashed-dotted line for sequences with correlation coefficient = 0.2.

the displacement and one for the acceptance test) so that the algorithm is potentially sensitive to correlations between successive quadruples of numbers. But it can also be sensitive to correlations of u_i with u_{i+4N} since one usually goes through the particles in order, causing $u_i, u_{i+4N}, ...$ to be used for the same purpose on the same particle. Unfortunately, the usual linear congruential generator has correlations between numbers separated by distances that are powers of 2; so it is not a good idea to simulate systems where N is a power of 2 with this generator. In general each Monte Carlo algorithm is sensitive to particular types of correlations, making it hard to define a universal test.

B. Reproducibility:

In the early days of computers, it was suggested that one could make a special circuit element which would deliver truly random numbers. Computer vendors have not supplied such an element because it is not trivial to design a device to deliver high quality numbers at a sufficient rate. Even more importantly, debugging codes would becomes much more difficult if each time the code was run a completely irreproducible sequence were to be generated. In Monte Carlo simulations, bugs may be manifest only at certain times, depending on the sequence of random numbers obtained. In order to detect the errors, it is necessary to repeat the calculations to find out how the errors occurred. The feature of reproducibility is also helpful while porting the program to a different machine. If we have a sample run from one machine available, then we can try an identical run on a different machine and verify that it ported correctly. Such reproducible random number generators are said to be *pseudo random* (PRNG), and we shall call the numbers produced by such generators as *PRN*s.

There is a conflict between the requirements of reproducibility and randomness. On a finite memory computer, at any step k in the sequence the PRNG has an internal *state* specifiable conceptually by an integer S_k, where the size of this integer is not necessarily related to the word length of the computer. For each state S in the sequence, there is a mapping that gives a random number the user sees, $u_k = F(S_k)$. We also have an iteration process to determine the next state of the sequence from the current state, $S_{k+1} = T(S_k)$. All PRNGs can be classified by the internal state space, the mapping, and the iteration. The sequence is defined once we have specified the initial starting state S_0 known as the *seed*. Fig. 3 illustrates the procedure for obtaining pseudo-random sequences described above.

Now let us return to the possible conflict between the properties of reproducibility and randomness: if it is reproducible then it cannot be perfectly random since knowing the sequence will make betting on the next number easy. How do we resolve the incompatibility between the two properties? In common sense terms, we mean a good PRNG is one whose numbers are uncorrelated as long as you do not explicitly try to back out the mapping and iteration processes and use that to predict another member of the sequence. PRNG's have not been designed to be good cryptographic sequences.

C. Speed:

It is of course desirable to generate the random numbers fast. While for some applications the generation of the random numbers is the limiting factor, many generators take only a few

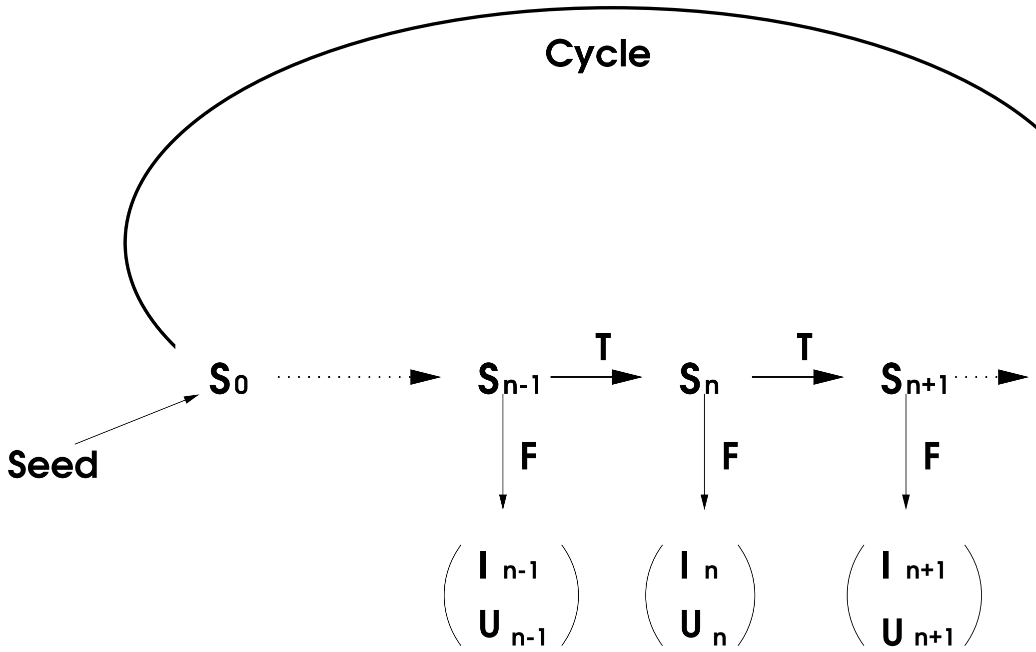

Figure 3: A pseudo-random sequence is defined by the internal state space, the mapping, the iteration, and the initial state.

clock cycles to deliver a new number so that usually the generation of random numbers is only a small fraction of the time required for the calculation. Hence speed is not a major concern unless, either the generator is extremely slow, or the remainder of the algorithm is extremely fast such as with a lattice spin model.

D. Large cycle length:

A pseudo random number generator is a finite state machine with at most 2^p different states where p is the number of bits that represent the state. One can easily see that the sequence must repeat after at most 2^p different numbers have been generated. The smallest number of steps after which the generator starts repeating itself is called the *period* or cycle length, L. Assuming all cycles have the same length, the number of disjoint cycles (i.e. having no states in common) is then: $2^p/L$.

A computer in 1997 might deliver 10^8 numbers/processor/ second (or 2^{26}). Hence it will take 1 second to exhaust a generator with a 26 bit internal state and 1 year to exhaust one with 2^{51} internal states. This suggests that it could be dangerous to use the 32 bit generators developed for the microprocessors of the 1980's on today's computers. After the sequence is exhausted, the "true" error of a simple MC evaluation integral will no longer decrease and one can be mislead into trusting an incorrect answer.

However, for many applications a small period will not in itself bias the results significantly. For example in MCMC we can think of the "state" of the random walk as consisting both of the coordinates of the particles (say $3N$ position variables) and of the internal state of the PRNG. The walk will repeat itself only if all the coordinates are exactly the same. Hence even if the random number sequence repeats, the particles will have moved on and have a different internal state. However, it is not a good idea to have repeating sequences, especially since it is easy to avoid.

Parallelization:

We next mention the implications of correlation and cycle length on parallel random number generators (PPRNG).

In order to get higher speed, Monte Carlo applications make extensive use of parallel computers, since these calculations are particularly well suited to such architectures and often require very long runs. A common way to parallelize Monte Carlo is to put identical "clones " on the various processors; only the random number sequences are different. It is therefore important for the sequences on the different processors to be uncorrelated. That is, given an initial segment of the sequence on one process, and the random number sequences on other processes, we should not be able to predict the next element of the sequence on the first process. For example, it should not happen that if we obtain random numbers of large magnitude on one process, then we are more likely to obtain large numbers on another.

Consider the following extreme case to demonstrate the impact of correlations. Suppose we perform identical calculations on each process, expecting different results due to the presence of a different random number sequence. If, however, we use the same sequence on each process, then we will get an identical result on each process and the the power of the parallel computer is wasted. Even worse we may incorrectly believe that the errors are much reduced because all processes give identical results. Such cases routinely occur when users first port their MC codes to parallel computers without considering how the random number sequence is to be parallelized.

Even if the correlations across processes are not perfect, any correlation can affect the random walk. It is generally true that inter-processor correlation is less important that intra-processor correlation, but that can depend on the application. The danger is that a particular parallel application will be sensitive to a particular correlation. New statistical tests have to be invented for correlation between processors.

The desire for *reproducibility*, when combined with *speed*, is also an important factor, and limits the feasible parallelization schemes. We shall next describe some common schemes for creating PPRNGs along with their merits.

1. **Central Server**

   One can maintain one particular process that serves as a centralized random number generator for all the processes. Any process that requires a random number obtains it from that process by sending messages. Such a scheme reduces the speed greatly since inter-processor communication is very expensive and the process needing the PRN must have exclusive access to the server to ensure that there are no conflicts. It also hinders reproducibility because the different processes may request random numbers in different orders in different runs of the program, depending on the network traffic and implementation of the communication software.

2. **Cycle Division**

   In one popular scheme the same iteration process is used on the different processes, but with widely separated seeds on each process. There are two related schemes: (i) the *leap frog* method where processor i gets $u_i, u_{i+M}, \ldots$, where M is the total number of processes. For example, process 1 gets the first member of the sequence, process 2 the second and so forth. (ii) In the *cycle splitting* method for M processors, process $i+1$ gets $u_{iL/M}, u_{iL/M+1}, ...$, where l is the cycle length. That is, the first process will get the first L/M numbers, the second process the second L/M numbers, and so forth.

Both methods require a fast way of advancing the PRNG a few steps; faster than iterating the sequence that number of steps. In the first method we need to be able to advance by M steps at each iteration. In the second method we need to be able to advance by L/M steps during initialization.

Statistical tests performed on the original sequence are not necessarily adequate for the divided sequence. For example, in the leap frog method correlations M numbers apart in the original sequence become adjacent correlations of the split sequence.

Clearly either method reduces the period of the original sequence by the number of processes. For example, with 512 nodes running at 100 MFlops one will exhaust the sequence of a PRNG with a 46 bit internal state (the common real*8 or long rng) in only 23 minutes! If the number of random numbers consumed is greater than expected, then the sequences on different processes could overlap.

3. **Cycle Parameterization**

   Another scheme makes use of the fact that some PRNGs have more than one cycle. If we choose the seeds carefully, then we can ensure that each random sequence starts out in a different cycle, and so two sequences will not overlap. Thus the seeds are parameterized (that is, sequence i gets a seed from cycle i, the sequence number being the parameter that determines its cycle). This is the case for the Lagged Fibonacci Generator described in the next section.

4. **Parameterized Iteration**

   Just as the disjoint cycles can be parameterized, so can many of the iteration functions for the internal state as with the Generalized Feedback Shift Register described in the next section. Here, sequence i gets iteration function T_i.

It is difficult to ensure reproducibility if the number of processors changes between one run and the next. This problem cannot be solved by the PPRNG in itself. In order to write a parallel MC application which gives identical results with a variable number of processors, it is necessary to write in terms of "virtual processors", each virtual processor having its own PRNG. Each physical processor would handle several virtual processors. It is unlikely that many programmers would go to this much trouble just to ensure that their code has this degree of portability unless it can be done automatically.

There are other consequences of the desire for reproducibility and speed. As an example, consider the simulation of a branching process. Suppose "neutron" paths are generated based on the outcome of various interactions between the neutrons and a medium. During a neutron flight, the neutron may collide with an atom and produce new neutrons (fission) or be absorbed (fusion). Efficient utilization of the processors requires good load balancing, and so one can move the computation of the statistics for new neutrons to different processors in order to keep the work evenly balanced. To ensure reproducibility in different runs, each neutron must be given a different random number sequence in a deterministic fashion. Thus, in a repetition of a run, even if the neutron is migrated to a different processor, the same random number sequence will be produced in determining its path. We also need to ensure that the random number sequence produced for each neutron is unique without incurring inter-processor communication. This can be accomplished by developing the ability to "spawn" unique sequences from an existing one.

We take the model that when a new process forks, a new sequence is generated for that process. Each sequence can be identified by a parameter if we parallelize the generators by any of the methods of parameterization described earlier. We ensure uniqueness of the new sequence by assigning each process a set $\mathcal{P}$ of parameters available for spawning. When a process forks, it partitions the elements of $\mathcal{P}$ among itself and its children, to create new sets of parameters available for spawning as shown in Fig. 4. Since the sets available for spawning on each process are disjoint, different sequences are obtained on each process. There is, however, the risk that eventually all the parameters could be used up; so we should not spawn too often. In the next section we will discuss some generators with very large numbers of possible parameters, so that quite a lot of spawning could be done before we would be in danger of repeating parameters.

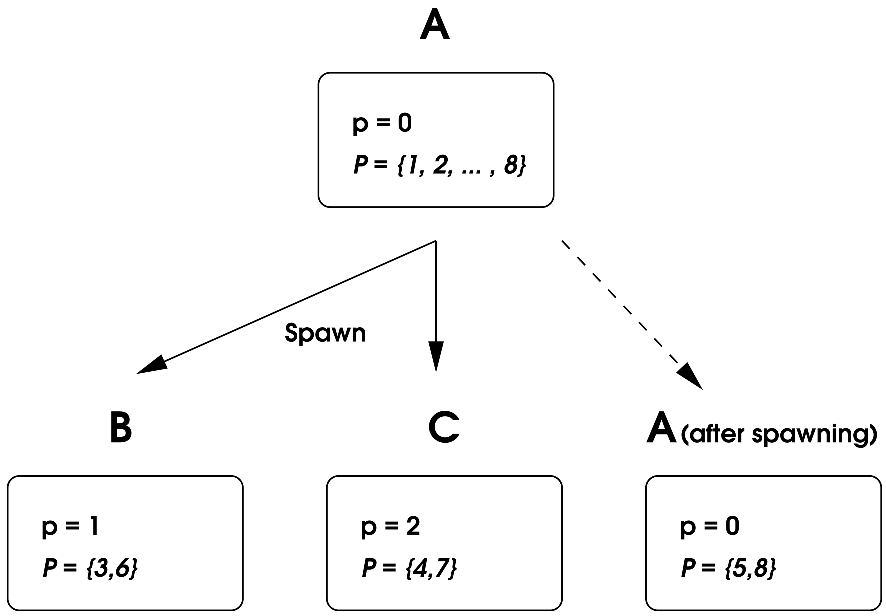

Figure 4: Process A spawns processes B and C. Each new process gets a new random number sequence parameterized by p and a set of parameters $\mathcal{P}$ for spawning.

On today's parallel computers communication is very slow compared to floating point performance. It is sometimes possible to use multiple random number sequences to reduce the communication costs in a Monte Carlo simulation. An example occurs in our parallel path integral calculations [19]. The total configuration space of the imaginary time path is divided into subspaces based on the "imaginary time" coordinate so that a given processor has control over a segment of imaginary time. Most moves are made on variables local to each processor but occasionally there are global moves to be made which move all values of imaginary time, where each processor has to evaluate the effect of the move on its variables. For these moves the number of messages can be cut in half by having available two different sequences of random numbers on each processor: (i) a local sequence which is different on each processor (and uncorrelated) and (ii) a global sequence shared by all processors. The global sequence permits all the processors to "predict" without communication what the global move will be, evaluate its consequence on their variables, and then "vote" whether that global move is acceptable. Thus half of the communication and synchronization cost is avoided compared to the scheme whereby a master processor would generate the global move, send it out to all the workers and tally up the results from the various processors.

3 Methods for Random Number Generation

We describe in this section some popular basic generators and their parallelization. We also mention about "combined generators", which are obtained from the basic ones.

3.1 Linear Congruential Generators

The most commonly used generator for pseudorandom numbers is the Linear Congruential Generator (LCG) [20]:

$$x_n = a x_{n-1} + b \pmod{m}. \tag{1}$$

where m is the modulus, a the multiplier, and c the additive constant or addend. The size of the modulus constrains the period, and it is usually chosen to be either prime or a power of 2.

This generator (with m a power of 2 and $c = 0$) is the de facto standard included with FORTRAN and C compilers. One of the biggest disadvantages to using a power of 2 modulus is that the least significant bits of the integers produced by these LCGs have extremely short periods. For example, $\{x_n \pmod{2^j}\}$ will have period 2^j [7]. In particular, this means the the least-significant bit of the LCG will alternate between 0 and 1. Since PRNs are generated with this algorithm, some cautions to the reader are in order: (i) The PRN should not be split apart to make several random numbers since the higher order bits are much more random than the lower order bits. (ii) One should avoid using the power of 2 modulus in batches of powers of 2. (For example if one has 1024 particles in 3 dimensions, one is using the PRNs 4096 at a time and the correlations between a PRN and one 4096 later may be large.) (iii) Generators with large modulus are preferable to ones with small modulus. Not only is the period longer, but the correlations are much less. In particular one should not use 32 bit modulus for careful work. In spite of this known defect of power of 2 LCGs, 48 bit multipliers (and higher) have passed many very stringent randomness tests.

Generally LCGs are best parallelized by parameterizing the iteration process, either through the multiplier or the additive constant. Based on the modulus, different parameterizations have been tried.

3.1.1 Power of 2 Modulus

The parameterization chooses a set of additive constants $\{b_j\}$ that are pairwise relatively prime, i.e. $\gcd(b_i, b_j) = 1$ when $i \neq j$ so that the sequences are generated in different orders. The best choice is to let b_j be the jth prime less than $\sqrt{m/2}$ [14]. One important advantage of this parameterization is that there is an inter-stream correlation measure based on the spectral test that suggests that there will be good inter-stream independence.

3.1.2 Prime Modulus

When the modulus m is prime, (usually very close to a power of 2 such as a Mersenne prime, $2^m - 1$, so that the operation of taking the modulus will be faster) a method based on using the multiplier, a, as the parameter to generate many sequences has been proposed. We start with a reference value of a and choose the multiplier for the jth stream as $a_j = a^{\ell_j} \pmod{m}$ where ℓ_j is the jth integer relatively prime to $m - 1$. This is closely related to the leapfrog method method discussed earlier. Conditions on a and efficient algorithms for computing ℓ_j can be found in a recent work of one of the authors [16].

The scheme given above can be justified based on exponential sums, which is explained in section 4.1. 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 inter-stream correlation also ensure good intra-stream independence via the spectral test?

3.2 Shift-Register Generators

Shift Register Generators (SRGs) [21, 22] are of the form:

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

where the x_n's and the a_i's are either 0 or 1. The maximal period of $2^k - 1$ and can be achieved using as few as two non-zero values of a_i. This leads to a very fast random number generator.

There are two ways to make pseudorandom integers out of the bits produced by Eq. (2). The first, called the digital multi-step method, takes n successive bits from Eq. (2) to form an integer of n-bits. Then n more bits are generated to create the next integer, and so on. The second method, called the generalized feedback shift-register, creates a new n-bit pseudorandom integer for every iteration of Eq. (2). This is done by constructing the n-bit word from the last bit generated, x_{n+k}, and $n-1$ other bits from the k bits of SRG state. Thus a random number is generated for each new bit generated. While these two methods seem different, they are very related, and theoretical results for one always hold for the other. Serious correlations can result if k is small. Reader's interested in more general information on SRGs should consult the references: [23, 21, 22].

The shift register sequences can be parameterized through the choice of a_i. One can systematically assign the values of a_i to the processors to produce distinct maximal period shift register sequences [24]. This scheme can be justified based on exponential sum bounds, as in the case of the prime modulus LCG. This similarity is no accident, and is based on the fact that both generators are maximal period linear recursions over a finite field [25].

3.3 Lagged-Fibonacci Generators

The Additive Lagged-Fibonacci Generator (ALFG) is:

$$x_n = x_{n-j} + x_{n-k} \pmod{2^m}, \quad j < k. \tag{3}$$

In recent years the ALFG has become a popular generator for serial as well as scalable parallel machines because it is easy to implement, it is cheap to compute and it does well on standard statistical tests [11], especially when the lag k is sufficiently high (such as $k = 1279$). The maximal period of the ALFG is $(2^k - 1)2^{m-1}$ [26, 27] and has $2^{(k-1)\times(m-1)}$ different full-period cycles [28]. Another advantage of the ALFG is that one can implement these generators directly in floating-point to avoid the conversion from integer to floating-point that accompanies the use of other generators. However, some care should be taken in the implementation to avoid floating point round-off errors [15].

In the previous sections we have discussed generators that can be parallelized by varying a parameter in the underlying recursion. Instead the ALFG can be parameterized through its initial values because of the tremendous number of different cycles. We produce different streams by assigning each stream a different cycle. An elegant seeding algorithm that accomplishes this is described in reference [28].

An interesting cousin of the ALFG is the Multiplicative Lagged-Fibonacci Generator (MLFG). It is defined by:

$$x_n = x_{n-j} \times x_{n-k} \pmod{2^m}, \quad j < k. \tag{4}$$

While this generator has a maximal-period of $(2^k - 1)2^{m-3}$, which is a quarter the length of the corresponding ALFG [27], it has empirical properties considered to be superior to ALFGs [11]. Of interest for parallel computing is that a parameterization analogous to that of the ALFG exists for the MLFG [29].

3.4 Inversive Congruential Generators

An important new type of PRNG that, as yet, has not found any widely distributed implementation is the Inversive Congruential Generator (ICG). This generator comes in two versions, the recursive ICG [30, 31]

$$x_n = a\overline{x}_{n-1} + b \pmod{m}, \tag{5}$$

and the explicit ICG [32]

$$x_n = \overline{an + b} \pmod{m}. \tag{6}$$

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 [33, 34]. Unfortunately the cost of doing modular inversion is considerable: it is $O(\log_2 m)$ times the cost of multiplication.

3.5 Combination Generator

Better quality sequences can often be obtained by combining the output of the basic generators to create a new random sequence as follows:

$$z_n = x_n \odot y_n \tag{7}$$

where $\odot$ is typically either the exclusive-or operator or addition modulo some integer m, and x and y are sequences from two independent generators. It is best if the cycle length of the two generators is relatively prime, for this implies that the cycle length of z will be the product of that of the basic generators. One can show that the statistical properties of z are no worse than those of x or y [11]. In fact, one expects it would be much superior but little has yet been proven to date.

Good combined generators have been developed by L'Ecuyer [35], based on the addition of Linear Congruential sequences.

4 Testing Random Number Generators

We have so far discussed the desired features of random number generators and described some of the popular generators used in Monte Carlo applications. Now the question arises: How good are the random number generators? While most generators are quite good for a variety of applications, there have been a few applications in which popular generators have been found to be inadequate [3, 4, 5, 6].

Sophisticated Monte Carlo applications often spend only a small fraction of their time in the actual random number generation. Most of the time is taken up by other operations. It is therefore preferable if we can test the random number generator fast, without actually using it in the application, to determine if it is good. For example in the *runs test* we divide the random number sequence into blocks that are monotonically increasing, and determine the distribution of the lengths of each block. This is a good test of randomness because it is fast and looks at correlations between longer and longer groups of random numbers. (Using a sequence of length 10^{12} we expect to see runs as long as 15.)

However, a single statistical test is not adequate to verify the randomness of a sequence, because typical MCMC applications can be sensitive to various types of correlations. However, if the generator passes a wide variety of tests, then our confidence in its randomness increases. The tests suggested by Knuth [7] and those implemented in the DIEHARD package by Marsaglia [36] are a standard. Since there are many generators which pass these tests there is no reason to consider one that is known to fail. Of course, any generator will eventually fail most tests, so we always must state how many numbers were used in the test. level of accuracy).

The second type of test is to run an application that uses random numbers in a similar manner to your applications, but for which the exact answer is known. For statistical mechanical applications, the two-dimensional Ising model (a simple lattice spin model) is often used since the exact answer is known and it has a phase transition so one expects sensitivity to long range correlations. There are several different MCMC algorithms that can be used to simulate the Ising model and the random numbers enter quite differently in each algorithm. Thus this application is very popular in testing random number generators and has often detected defects in generators. We can test parallel generators on the Ising model application by assigning distinct random number sequences to subsets of lattice sites [37].

How good are the popular generators on the Ising model? Ferrenberg, et al [3] found that certain generators such as the particular shift register generator they used (R250) failed with the Wolff algorithm, while the simple and much maligned 32 bit LCG called CONG did well.

Similar defects in popular generators have also been observed by others [4, 5, 6]. However, with the Metropolis algorithm, CONG performed much worse than R250. Thus it appears that it is the combination of problem *and* generator that must be considered in choosing a suitable generator. Thus statistical tests are not adequate; we must also test the generators in our actual application.

Of course in almost all cases we do not know the exact answer. How then can we trust a given generator on our problem? The only general approach is to run our application with several different types of generators that are known to have good statistical properties. Hopefully at least two will agree within the error bars and they can be used on further similar runs with that algorithm. A word of caution: the test runs must be at least as long as the production runs because subtle correlations in the PRNG may not show up until long runs are made. This approach is quite computationally expensive and if done seriously would increase the cost of all MC simulations by a factor of two. (If you decide at the end that all generators are good, you can recover the time by averaging the results together.)

4.1 Parallel Tests

We now mention some tests of parallel random number generators.

Exponential sums: The exponential sum (Fourier transform of the density) of a sequence $u_0, ..., u_k$ is:

$$C_g(k) = \sum_{j=0}^{k-1} e^{2\pi i x_j g}. \tag{8}$$

For a random sequence $\langle |C_g(k)|^2 \rangle = k$ (for $g \neq 0$). This fact can be used to test correlation within, and between, random number sequences [13, 16]. Consider two random sequences X and Y and define the exponential sum cross-correlation:

$$C_g(h, l, k) = \sum_{j=0}^{k-1} e^{2\pi i g (x_{h+j} - y_{l+j})} \tag{9}$$

In each term of this sum, we find the difference between an element of each sequence at a fixed offset apart. If this difference were uniformly distributed, then we should have: $\langle |C(j, l, k)|^2 \rangle = k$

Parallel spectral test: Percus and Kalos [14] have developed a version of the spectral test for parallel linear congruential generators.

Interleaved tests: We create a new random sequence by interleaving several random sequences, and test this new sequence for randomness using standard sequential tests.

Fourier transform test: Generate a two dimensional array of random numbers with each row in the array consisting of consecutive random numbers from one particular sequence. The two dimensional Fourier transform can be performed. For a truly uncorrelated set of sequences the coefficients (except the constant term) should be close to 0.

Blocking test: In the blocking test we add random numbers from several streams as well as from within a stream. If the streams are independent, then the distribution of these sums will approach the normal distribution.

We now give some test results for the LCG with (parameterized) prime addend and a modified version of the LFG. Both of these generators performed acceptably in the sequential tests with 10^{11} random numbers. Preliminary results from other tests of PPRNG can also be found in the paper by Coddington [37].

We first tested the generators by interleaving about 1000 pairs of random sequences each containing about 10^8 PRNs. Both the generators passed this tests. Next we simulated parallelism on the 16×16 Ising model by using a different random sequence for each lattice site in the Metropolis algorithm. The LCG (with identical seeds) failed badly as can be seen from the dashed line in Fig. 5. We then generalized the statistical tests, interleaving 256 sequences at a time. The LCG again failed, demonstrating the effectiveness of these tests in detecting non-random behavior. The modified LFG passed these tests.

In the tests mentioned above, we had started each parameterized LCG with the same seed but used different additive constants. Even when we discarded the first million random numbers from each sequence, the sequences were still correlated. However, when we started the streams from different, systematically spaced seeds, the LCG passed all statistical tests (including parallel ones) with up to 10^{10} - 10^{12} numbers. We finally repeated the parallel

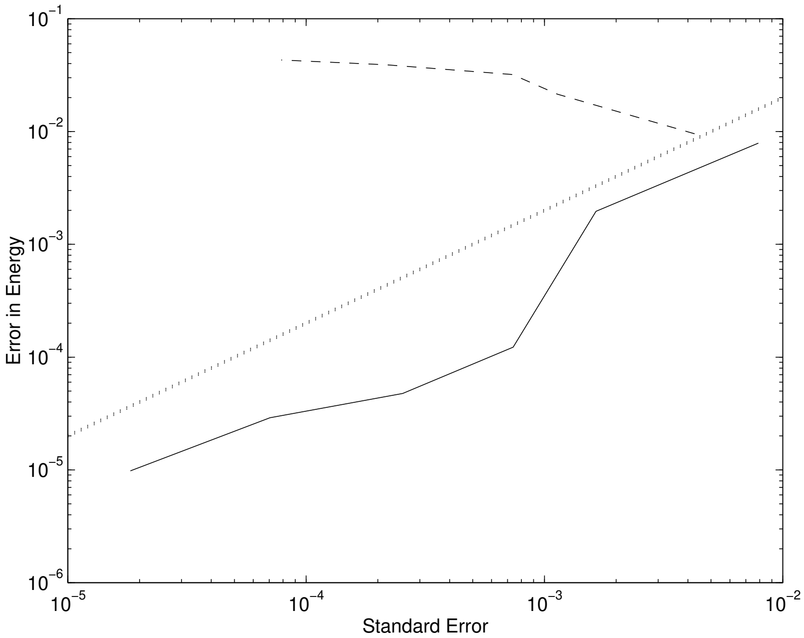

Figure 5: Plot of the actual error versus the internally estimated standard deviation of the energy error for Ising model simulations with the Metropolis algorithm on a 16×16 lattice with a different Linear Congruential sequence at each lattice site. The dashed line shows the results when all the Linear Congruential sequences were started with the same seeds but with different additive constants. The solid line shows the results when the sequences were started with different seeds. We expect around 95% of the points to be below the dotted line (which represents an error of two standard deviations) with a good generator.

Metropolis algorithm simulation with 10^{11} random numbers, and both generators performed acceptably, as can be seen from the solid line in Fig. 5.

Apart from verifying the quality of two particular generators, our results illustrate that it is commonplace for generators to pass some test and fail others. It is important to test parallel generators the way they will be seeded, since correlations in the seeding can lead to correlations in the resulting sequences.

5 Quasi-random numbers

It is well known that the error estimate from any MC calculations (or in general from a simulation) converges as $N^{-1/2}$ where N is the number of random samples or equivalently the computer time expended. Recently there has been much research into whether non-random sequences could result in faster convergence, but still have the advantages of MC in being applicable to high-dimensional problems.

Consider the integral:

$$I = \int_0^1 d^s x f(x) \tag{10}$$

The Monte Carlo estimate for this integral is:

$$\hat{I}_n = \frac{1}{n}\sum_{i=1}^{n} f(x_i), \tag{11}$$

where the points x_i are to be uniformly distributed in the s-dimensional unit cube. An important, and very general, result that bounds the absolute error of the Monte Carlo estimate is the Koksma-Hlwaka inequality [25]:

$$|I - \hat{I}| \leq D_s(x_i)V(f), \tag{12}$$

where $D_s(x_i)$ is the discrepancy of the points x_i as defined by Eq. (13), and $V(f)$ is the total variation of f on $[0,1)^s$ in the sense of Hardy and Krause [38]. The total variation is roughly the average absolute value of the s^{th} derivative of $f(x)$.

Note that Eq. (12) gives a deterministic error bound for integration because $V(f)$ depends only on the nature of the function. Similarly, the discrepancy of a point set is a purely geometric property of that point set. When given a numerical quadrature problem, we must cope with whatever function we are given, it is really only the points, x_i, that we control. Thus one approach to efficient integration is to seek point sets with small discrepancies. Such sets are necessarily not random but are instead referred to as *quasi-random numbers* (QRNs).

The s dimensional discrepancy of the N points, x_i is defined as:

$$D_s(x_i) = \sup_B \left| \frac{\#(B, x_i)}{N} - \lambda(B) \right|. \tag{13}$$

Here B is any rectangular subvolume inside the s-dimensional unit cube with sides parallel to the axes and volume $\lambda(b)$. The function $\#(B, x_i)$ counts the number of points of x_i in B. Hence a set of points with a low discrepancy covers the unit cube more uniformly than one with a large discrepancy. A lattice has a very low discrepancy, however adding additional points to the lattice is slow in high dimensions. The discrepancy of a lattice does not decrease smoothly with increasing N.

A remarkable fact is that there is a lower (Roth) bound to the its discrepancy [39]:

$$D_s^*(x_i) \geq C_s N^{-1} (\log N)^{(s-1)/2}, s > 3. \tag{14}$$

This gives us a target to aim at for the construction of low-discrepancy point sets (quasi-random numbers). For comparison, the estimated error with random sequences is:

$$\left\langle (I - \hat{I})^2 \right\rangle = \sqrt{\frac{1}{N} \left[\int f^2 - (\int f)^2 \right]}. \tag{15}$$

The quantity in brackets, the variance, only depends on f. Hence the standard deviation of the Monte Carlo estimate is $O(N^{-1/2})$. This is much worse than the bounds of Eq. (14) as a function of the number of points. It is this fact that has motivated the search for quasi-random points.

What goes wrong with this argument as far as most high-dimensional problems in physical science is that the Koksma-Hlwaka bound is extremely poor for large s. Typical integrands become more and more singular the more one differentiates. The worse case is the Metropolis rejection step: the acceptance is itself discontinuous. Even assuming $V(f)$ were finite, it is likely to be so large (for large s) that truly astronomical values of N would be required for the bound in Eq. (14) to be less than the MC convergence rate given in Eq. (15).

We will now present a very brief description of quasi-random sequences. Those interested in a much more detailed review of the subject are encouraged to consult the recent work of Niederreiter [25]. An example of a one-dimensional set of quasi-random numbers is the van der Corput sequence. First we choose a base, b, and write an integer n in base b as $n = \sum_{i=0}^{\log_b n} a_i b^i$. Then we define the van der Corput sequence as $x_n = \sum_{i=0}^{\log_b n} a_i b^{-i-1}$. For base $b = 3$, the first 12 terms of the van der Corput sequence are:

$$\left\{0, \frac{1}{3}, \frac{2}{3}, \frac{1}{9}, \frac{4}{9}, \frac{7}{9}, \frac{2}{9}, \frac{5}{9}, \frac{8}{9}, \frac{1}{27}, \frac{10}{27}, \frac{19}{27}, \dots\right\}. \tag{16}$$

One sees intuitively how this sequence, while not behaving in a random fashion, fills in all the holes in a regular and low-discrepancy way.

There are many other low-discrepancy point sets and sequences. Some, like the Halton sequence [40] use the van der Corput sequence. There have been many others which are thought to be more general and have provably smaller discrepancy. Of particular note are the explicit constructions of Faure [41] and Niederreiter [42].

The use of quasi-random numbers in quadrature has not been widespread because the claims of superiority of quasi-random over pseudo-random for quadrature have not been shown empirically especially for high dimensional integrals and never in MCMC simulations [43]. QRNs work best in low dimensional spaces where the spacing between the points can be made small with respect to the curvature of the integrand. QRNs work well for very smooth integrands. The Boltzmann distribution $\exp\left(-V(R)/k_B T\right)$ is highly peaked for R a many dimensional point. Empirical studies have shown that the number of integration points needed for QRN error to be less than the MC error increases very rapidly with the dimensionality. Until the asymptotic region is reached, the error of QRN is not very different from simple MC. Also there are many MC tricks which can be used to reduce the variance such as importance sampling, antithetic sampling and so forth. Finally MC has a very robust way of estimating errors; that is one of its main advantages as a method. Integrations with QRNs have to rely on empirical error estimates since the rigorous upper bound is exceedingly large. It may happen that the integrand in some way correlates with the QRN sequence so that these error estimates are completely wrong, just as sometimes happens with other deterministic integration methods.

There seem to be some advantages of quasi-random over pseudo-random for simple smooth integrands that are effectively low dimensional (say less than 15 dimensions), but they are much smaller than one is lead to expect from the mathematical results.

ACKNOWLEDGMENT

We wish to acknowledge the support of DARPA through grant number *DABT 63-95-C-0123* and the NCSA for computational resources.

References

[1] P. A. M. Dirac. *Proc. R. Soc. London Ser A*, 123:734, 1929.

[2] J. B. Anderson. In S. R. Langhoff, editor, *Understanding Chemical Reactivity.* Kluwer, Dordrecht, The Netherlans, 1995.

[3] A. M. Ferrenberg, D. P. Landau, and Y. J. Wong. Monte Carlo simulations: Hidden errors from "good" random number generators. *Phys. Rev. Let.*, 69:3382–3384, 1992.

[4] P. Grassberger. On correlations in 'good' random number generators. *Phys. Lett. A*, 181(1):43–46, 1993.

[5] W. Selke, A. L. Talapov, and L. N. Schur. Cluster-flipping Monte Carlo algorithm and correlations in "good" random number generators. *JETP Lett.*, 58(8):665–668, 1993.

[6] F. Schmid and N. B. Wilding. Errors in Monte Carlo simulations using shift register random number generators. *Int. J. Mod. Phys. C*, 6(6):781–787, 1995.

[7] D. E. Knuth. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Second edition.* Addison-Wesley, Reading, Massachusetts, 1981.

[8] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in FORTRAN*. Cambridge University Press, New York, NY, second edition, 1994.

[9] S. K. Park and K. W. Miller. Random number generators: good ones are hard to find. *Comm. of the ACM*, 31:1192–1201, 1988.

[10] P. L'Ecuyer. Random numbers for simulation. *Comm. of the ACM*, 33:85–97, 1990.

[11] G. Marsaglia. A current view of random number generators. In *Computing Science and Statistics: Proceedings of the XVIth Symposium on the Interface*, pages 3–10, 1985.

[12] P. Coddington. Random number generators for parallel computers, 28 April 1997. http://www.npac.syr.edu/users/paulc/papers/NHSEreview1.1/PRNGreview.ps.

[13] M. Mascagni, S. A. Cuccaro, D. V. Pryor, and M. L. Robinson. Recent developments in parallel pseudorandom number generation. In D. E. Keyes, M. R. Leuze, L. R. Petzold, and D. A. Reed, editors, *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, volume II, pages 524–529, Philadelphia, Pennsylvania, 1993. SIAM.

[14] O. E. Percus and M. H. Kalos. Random number generators for MIMD parallel processors. *J. of Par. Distr. Comput.*, 6:477–497, 1989.

[15] R. P. Brent. Uniform random number generators for supercomputers. In *Proceedings Fifth Australian Supercomputer Conference*, pages 95–104. 5ASC Organizing Committee, 1992.

[16] M. Mascagni. Parallel linear congruential generators with prime moduli. *IMA Reprint 1470 and submitted*, 1997.

[17] M. Mascagni S. A. Cuccaro 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*, pages 279–284, Philadelphia, Pennsylvania, 1995. SIAM.

[18] M. Kalos and P. Whitlock. *Monte Carlo Methods.* Wiley-Interscience, New York, 1986. Volume I: Basics.

[19] D. M. Ceperley. Path integrals in the theory of condensed helium. *Reviews of Modern Physics*, 67(2):279–355, 1995.

[20] D. H. Lehmer. Mathematical methods in large-scale computing units. In *Proceedings of the 2nd Symposium on Large-Scale Digital Calculating Machinery*, pages 141–146, Cambridge, Massachusetts, 1949. Harvard University Press.

[21] T. G. Lewis and W. H. Payne. Generalized feedback shift register pseudorandom number algorithms. *J. of the ACM*, 20:456–468, 1973.

[22] R. C. Tausworthe. Random numbers generated by linear recurrence modulo two. *Math. Comput.*, 19:201–209, 1965.

[23] S. W. Golomb. *Shift Register Sequences.* Aegean Park Press, Laguna Hills, California, 1982. Revised Edition.

[24] J. L. Massey. Shift-register synthesis and bch decoding. *IEEE Trans. Information Theory*, 15:122–127, 1969.

[25] H. Niederreiter. *Random number generation and quasi-Monte Carlo methods.* SIAM, Philadelphia, Pennsylvania, 1992.

[26] R. P. Brent. On the periods of generalized Fibonacci recurrences. *Math. Comput.*, 63:389–401, 1994.

[27] G. Marsaglia and L.-H. Tsay. Matrices and the structure of random number sequences. *Linear Alg. and Applic.*, 67:147–156, 1985.

[28] M. Mascagni, S. A. Cuccaro, D. V. Pryor, and M. L. Robinson. A fast, high-quality, and reproducible lagged-Fibonacci pseudorandom number generator. *J. Comput. Physics*, 15:211–219, 1995.

[29] M. Mascagni. A parallel non-linear Fibonacci pseudorandom number generator, 1997. Abstract, 45th SIAM Annual Meeting.

[30] J. Eichenauer and J. Lehn. A nonlinear congruential pseudorandom number generator. *Statist. Hefte*, 37:315–326, 1986.

[31] H. Niederreiter. Statistical independence of nonlinear congruential pseudorandom numbers. *Montash. Math.*, 106:149–159, 1988.

[32] H. Niederreiter. On a new class of pseudorandom numbers for simulation methods. *J. Comput. Appl. Math.*, 56:159–167, 1994.

[33] G. Marsaglia. Random numbers fall mainly in the planes. *Proc. Nat. Acad. Sci. U.S.A.*, 62:25–28, 1968.

[34] G. Marsaglia. The structure of linear congruential sequences. In S. K. Zaremba, editor, *Applications of Number Theory to Numerical Analysis*, pages 249–285. Academic Press, New York, 1972.

[35] P. L'Ecuyer. Efficient and portable combined random number generators. *Comm. of the ACM*, 31:742–774, 1988.

[36] G. Marsaglia. Diehard. ftp://stat.fsu.edu/pub/diehard.

[37] P. Coddington. Tests of random number generators using Ising model simulations. *Int. J. of Mod. Phys. "C"*, 7(3):295–303, 1996.

[38] E. Hlwaka. Funktionen von beschränkter variation in der theorie der gleichverteiling. *Ann. Mat. Pura Appl.*, 54:325–333, 1961.

[39] K. F. Roth. On irregularities of distribution. *Mathematika*, 1:73–79, 1954.

[40] J. H. Halton. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numer. Math.*, 2:84–90, 1960.

[41] H. Faure. Using permutations to reduce discrepancy. *J. Comp. Appl. Math.*, 31:97–103, 1990.

[42] B. L. Fox P. Bratley and H. Niederreiter. Implementation and tests of low-discrepancy point sets. *ACM Trans. on Modeling and Comp. Simul.*, 2:195–213, 1992.

[43] W. J. Morokoff and R. E. Caflisch. Quasi-Monte Carlo integration. *J. Comp. Phys.*, 122:218–230, 1995.