Chapter 19
Generating Random Numbers

Hans Ekkehard Plesser

Abstract Large-scale parallel surrogate data generation and neuronal network simulation require far more pseudorandom numbers than many popular pseudorandom number generators (PRNGs) can deliver. Fortunately, random number generation has progressed from black art to science over the past two decades, providing us with reliable algorithms for generating random numbers and rigorous tests for such algorithms. In this chapter we will first review requirements for good PRNGs, before presenting the basic principles of some widely used generators, including cryptographic generators. We then discuss seeding strategies and the transformation of integer random numbers to random variates following other distributions.

Random number generation has progressed from black art to science over the past two decades (Panneton et al. 2006; Park and Miller 1988), providing us with reliable algorithms for generating random numbers and rigorous tests for such algorithms (L'Ecuyer and Simard 2007). No deterministic algorithm can generate truly random numbers. Strictly speaking, we are considering *pseudorandom number generators* (PRNGs). For any generator, there will be some application that reverberates with the correlation structure hidden in the generator's algorithm, leading to false results (Ferrenberg et al. 1992; Compagner 1995).

Before considering pseudorandom number generators in detail, let us briefly consider the number of random numbers required for a large project that keeps a 1024-core computer busy for a year, or roughly 2^{55} nanoseconds. Assuming a clock speed of 1 GHz, the project goes through $2^{65} \approx 4 \times 10^{19}$ clock cycles. If one pseudorandom number is consumed per 10^4 clock cycles, a total of 4×10^{15} numbers will be consumed (Knuth 1998, Chap. 3.6). Empirical evidence indicates

H.E. Plesser (✉)
Dept. of Mathematical Sciences and Technology, Norwegian University of Life Sciences, PO Box 5003, 1432 Aas, Norway
RIKEN Brain Science Institute, 2-1 Hirosawa, Wakoshi, Saitama 351-0198, Japan
e-mail: hans.ekkehard.plesser@umb.no
url: http://arken.umb.no/~plesser

S. Grün, S. Rotter (eds.), *Analysis of Parallel Spike Trains,*
Springer Series in Computational Neuroscience 7,
DOI 10.1007/978-1-4419-5675-0_19,

that a PRNG used in this project should be able to generate a sequence of at least $(4 \times 10^{15})^3 = 64 \times 10^{45}$ pseudorandom numbers before repeating itself (Brent 2006; Matsumoto et al. 2006; L'Ecuyer and Simard 2007). This exceeds by far the capabilities of many widely used generators.

In this chapter we will first review requirements for good PRNGs, before presenting the basic principles of some widely used generators, including cryptographic generators. We then discuss seeding strategies and the transformation of integer random numbers to random variates following other distributions.

For details, please see the textbook by Gentle (2003), Knuth's classical treatise (1998), or the review by L'Ecuyer and Simard (2007); the latter two references include a comprehensive treatment of tests for PRNGs. Devroye (1986) is the canonical reference for nonuniform random variates.

19.1 Requirements for Pseudorandom Number Generators

A pseudorandom number generator is an algorithm providing a *drawing function* $\mathcal{D}()$ which emits a new pseudorandom number $r_j \in \mathbb{W}$ each time the drawing function is executed; $\mathbb{W}$ is the finite, discrete set from which numbers are chosen. The *period* ρ of the generator is the number of PRNs that can be drawn before the sequence of numbers repeats itself. If different initializations of the generator can lead to different periods, one considers the minimal period as the period of the generator. For most modern generators, the period is much larger than the set of values $\mathbb{W}$.

Brent (2006) lists the following requirements for good PRNGs:

Uniformity The numbers $(r_0, r_1, \ldots)$ generated should be distributed uniformly over $\mathbb{W}$.

Independence Any subsequence of the full sequence $(r_0, r_1, \ldots)$ of random numbers should be statistically independent.

Long period From a PRNG with period ρ, one should draw at most $\rho^{1/3}$ random numbers (*cube-root rule*; Brent 2006; Matsumoto et al. 2006; L'Ecuyer and Simard 2007).

Initialization Any choice of seed should yield equally random numbers.

Skip ahead Certain random generators allow one to "skip ahead", e.g., to go directly to the 2^{100}th random number in a sequence (L'Ecuyer et al. 2002; Haramoto et al. 2008).

Repeatability Especially when debugging simulations, it is crucial that the same random number sequence can be generated over and over again.

Portability A PRNG should produce exactly the same sequence of numbers for any computer on which it is run, without modifications to the source code.

Efficiency Random number generators should consume as little as possible of the total simulation time.

Unpredictability A truly random number sequence is unpredictable: no matter how long we observe the sequence, we will never be able to predict the next number. This is not the case for most PRNGs.

Uniformity and independence are ascertained by standard tests for pseudorandom number generators, while the period is usually known from the design of the generator. Initialization has received systematic attention only recently (Matsumoto et al. 2007); we will return to it in Sect. 19.4.

The ability to skip ahead in a random number stream is essential for the parallel generation of random number sequences and thus useful for generating large numbers of surrogate spike trains. It will be discussed in detail in Sect. 19.4.1.

Since pseudorandom number generators are based on deterministic algorithms, it is straightforward to reproduce their output in repeated simulations, provided that their initial state is known. Ill-designed parallel programs may still lead to irreproducible consumption of random numbers; we will not discuss such issues here.

Portability and efficiency of PRNGs are of lesser concern today than a decade ago, since several well-tested, portable PRNG libraries exist. An interesting new development are PRNGs exploiting the superior performance of vector extensions to CPUs and of graphics processors (Saito and Matsumoto 2008; Tzeng and Wei 2008).

Unpredictability of PRNGs is of little concern in simulations, while it is essential in cryptography and gaming. We will discuss unpredictable cryptographic generators in Sect. 19.3.

19.2 Recurrence-Based Generators

Most pseudorandom number generators used today are recurrence-based generators. These generators are defined by three mathematical functions operating upon a state variable Σ. The random generator is initialized by a *seed* value S according to a *seeding rule*

$$\Sigma_0 = \mathcal{S}(S). \tag{19.1}$$

Each time a new random number is required, the state of the generator is updated by an *iteration rule*

$$\Sigma_j = \mathcal{I}(\Sigma_{j-1}) \quad (j > 0), \tag{19.2}$$

before the jth random number is drawn from the state variable according to a *drawing rule*

$$r_j = \mathcal{D}(\Sigma_j). \tag{19.3}$$

The size of the state variable Σ sets an upper limit for the period of the generator. If Σ is n bits long, then the generator is at all times in one of 2^n states. Since (19.2) dictates that each state has exactly one successor state, it follows that the longest possible sequence of states without repetition has 2^n states, limiting the period to $\rho \leq 2^n$. PRNGs with 32-bit integers as state variables thus cannot have periods of more than $\rho = 2^{32} \approx 10^9$, which is far too short to be of practical use.

The random numbers r_j that can be drawn from a generator are typically 32-bit integers, although some generators provide floating point numbers directly. The

range of values that can be generated stretches from 0, 1, or 2 to some large integer $M < 2^n$. Note that the period ρ of a generator can be much larger than M.

19.2.1 Linear Congruential Generators

Additive linear congruential generators (LCGs) have been popular due to their simplicity and efficiency and are infamous for their weaknesses (Knuth 1998, Chap. 3.2.1). Seeding and drawing functions are trivial for these generators, while the iteration function is straightforward modulo arithmetic:

$$\mathcal{S}(s) = s \quad (0 < s < m), \tag{19.4}$$

$$\mathcal{D}(\Sigma) = \Sigma, \tag{19.5}$$

$$\mathcal{I}(\Sigma) = (a\Sigma + c) \bmod m, \tag{19.6}$$

where a is the *multiplier*, c the *increment*, and m the *modulus*. Lewis et al. (1969) proved that choosing $a = 7^5 = 16807$, $c = 0$, and $m = 2^{31} - 1 = 2{,}147{,}483{,}647$ results in a generator with maximum period, i.e., period $\rho = 2^{31} - 2$. Care must be taken to avoid integer overflow when implementing this algorithm using 32-bit integers (Park and Miller 1988).

Even though this generator has been considered a "minimal standard" for a long time (Park and Miller 1988; Press et al. 1992, Chap. 7), it is no longer a viable choice for large-scale simulations today due to its short period: strict application of the cube-root rule would allow the use of only 1290 random numbers, and even a common PC will zip through the entire period within a minute.

19.2.2 Lagged Fibonacci Generators

Lagged Fibonacci generators (LFG) were first proposed by Mitchell and Moore some 50 years ago and have long been considered reliable (Knuth 1998, Chap. 3.2.2). Their state vector $\mathbf{\Sigma}$ consists of the last l 32-bit random numbers drawn according to the generalized Fibonacci recurrence relation

$$r_n = (r_{n-s} \pm r_{n-l}) \bmod m \quad (0 < s < l \leq n). \tag{19.7}$$

Several valid pairs (l, s) of long and short lags are given by Knut (1998, Chap. 3.2.2, Table 1). Mitchell and Moore originally proposed lags (55, 24), while Knuth recommends (100, 37) in his Chap. 3.6. For moduli of the form $m = 2^k$, the LFG generator has a period of $\rho = 2^{k-1}(2^l - 1)$. For the parameters given by Knuth, one thus obtains a period $\rho \approx 2^{129}$.

Some weaknesses in these generators can be overcome by *decimation*, e.g., using only the first 100 of every 1009 numbers generated (Lüscher 1994). This gives the

canonical implementation of the LFG. Source Code is freely available from Donald Knuth's website.[1]

Properly initializing the state vector $\mathbf{\Sigma}$, which contains l 32-bit integers, based on a single 32-bit seed S is not trivial; Knuth has revised the initialization routine for the LFG as recently as in 2002.[2] The present initialization scheme appears robust under stringent tests (Matsumoto et al. 2007).

L'Ecuyer (2004) advises strongly against the use of LFGs, since the triplets (X_n, X_{n-s}, X_{n-l}) have strong correlations between themselves, so that simulators which should happen to combine random numbers in these intervals in unfortunate ways may suffer.

Marsaglia and Tsang (2004) suggest to combine a lagged Fibonacci generator with lags 97 and 33 with an arithmetic sequence of the form $c - jd \bmod p$ for suitable constants c, d, and p, to obtain a generator with period $\rho \approx 2^{202}$. This generator does not suffer from the shortcomings of "plain" LFGs, and thus one does not need to decimate the random number stream.

19.2.3 Combined Multiple Recursive Generators

Multiple recursive generators (MRG) generalize the lagged Fibonacci generators by introducing more terms and arbitrary coefficients into the recurrence (L'Ecuyer 2004)

$$X_n = (a_1 X_{n-1} + a_2 X_{n-2} + \cdots + a_k X_{n-k}) \bmod m. \tag{19.8}$$

LCGs are a special case of an MRG for $k = 1$, while LFGs are MRGs for which two $a_j = \pm 1$ while all other coefficients vanish. Combining two such generators with carefully chosen parameters results in a combined multiple recursive generator (CMRG) such as `MRG32k3a` with the following equations describing the recurrence and drawing (L'Ecuyer 1999):

$$\Sigma_{1,n} = (1{,}403{,}580 \times \Sigma_{1,n-2} - 810{,}728 \times \Sigma_{1,n-3}) \bmod m_1, \tag{19.9}$$

$$\Sigma_{2,n} = (527{,}612 \times \Sigma_{2,n-1} - 1{,}370{,}589 \times \Sigma_{2,n-3}) \bmod m_2, \tag{19.10}$$

$$r_n = (\Sigma_{1,n} - \Sigma_{2,n}) \bmod m_1, \tag{19.11}$$

$$m_1 = 2^{32} - 209 = 4{,}294{,}967{,}087, \tag{19.12}$$

$$m_2 = 2^{32} - 22{,}853 = 4{,}294{,}944{,}443, \tag{19.13}$$

where Σ_1 and Σ_2 are stored as doubles with 53-bit mantissa. This combined generator has a period $\rho = (m_1^3 - 1)(m_2^3 - 1)/2 \approx 2^{191} \approx 10^{57}$ and passes all tests in the TestU01 suite (L'Ecuyer and Simard 2007). An implementation in C is available from L'Ecuyer.[3] It is included in Matlab 7.7 and later. The `cmrg` generator in

[1] http://www-cs-faculty.stanford.edu/~knuth/programs/rng.c.

[2] As of the 9th printing of Knuth (1998).

[3] http://www.iro.umontreal.ca/~lecuyer/myftp/papers/combmrg2.c.

the GNU Scientific Library is a similar CMRG with a period $\rho \approx 2^{185}$, which also passes all TestU01 tests but shows some weakness in initialization tests (Galassi et al. 2001; Matsumoto et al. 2007). We will return to `MRG32k3a` in Sect. 19.4.1.

19.2.4 Mersenne Twister and Related Generators

Mersenne Twister (MT) random number generators were introduced by Matsumoto and Nishimura (1998) based on a class of generators known as *feedback shift register* generators. These generators do not operate on integer or double numbers, but on vectors of individual bits. This permits fast code based on bit-shift and bit-wise logical operations, thorough mathematical analysis, and efficient skipping ahead. Mersenne Twisters have thus become very popular in the ten years since their invention. For an introduction to the principles behind Mersenne Twisters and related generators, see L'Ecuyer (2004), on which the following is based.

For MT generators, all arithmetic is done modulo 2, i.e., in the finite field $\mathbb{F}_2$ with the two elements 0 and 1. The state variable $\mathbf{\Sigma}$ is a vector of k 0s and 1s, and iteration and drawing rules are given by

$$\mathbf{\Sigma}_j = \mathbf{A}\mathbf{\Sigma}_{j-1}, \tag{19.14}$$

$$\mathbf{r}_j = \mathbf{B}\mathbf{\Sigma}_j, \tag{19.15}$$

$$u_j = \sum_{l=1}^{w} r_{j,l} 2^{-l}, \tag{19.16}$$

with the $k \times k$ transition matrix $\mathbf{A}$ and the $w \times k$ output transformation matrix $\mathbf{B}$, both with elements from $\mathbb{F}_2$. $u_j \in [0, 1)$ is the output of the algorithm given as a double. Since doubles have only a 53-bit mantissa, one usually has $w \leq 53$, while $k \gg w$. The matrix multiplications in the equations above can be implemented efficiently through bit-shift and bit-wise logical operations.

The most commonly used Mersenne Twister today is MT19937 published in 1998, with an improved initialization routine published in 2002; source code is available from Matsumoto and others[4] and is available in the GNU Scientific Library as `mt19937`[5]; it is the default PRNG in NumPy and Matlab (since version 7.4). MT19937 has $k = 19{,}937$, i.e., the transition matrix $\mathbf{A}$ is a very sparse matrix of $19{,}937 \times 19{,}937$ bits. The state variable $\mathbf{\Sigma}$ correspondingly contains 19,937 bits, stored in 624 32-bit words. The period of the generator is $\rho = 2^{19{,}937} - 1 \approx 10^{6{,}000}$, so that even after application of the cube-root rule we are left with a mindboggling number of usable random numbers. MT19937 is fast in spite of its huge state vector and has passed all tests in the TestU01 testsuite except two, which all feedback shift register generators fail. On this failure, L'Ecuyer and Panneton (2005) comment that

[4] http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html.

[5] GSL v. 1.2 and later; earlier versions have the defective pre-2002 initialization routine.

"this is very unlikely to cause a problem in practice, unless the system we simulate has a lot to do with linear dependencies among bits". Doubts about the MT19937 recently raised by Kim et al. (2008) were based on a faulty analysis (Plesser and Jahnsen 2010).

19.2.5 Nonlinear Random Number Generators

All PRNGs presented so far are linear. Nonlinear generators, such as inversive congruential generators (ICG) proposed first by Eichenauer and Lehn (1986), are based on recurrences of the type

$$X_{j+1} = (a\overline{X}_j + b) \bmod m,$$

where the bar denotes the inverse with respect to multiplication modulo m, and m is a large prime. Integer division modulo a large prime is slow, whence ICGs are about 50 times slower than MT19937 (L'Ecuyer and Simard 2007). Due to their short period (typically $\rho \approx 2^{31}$), ICGs also fail the TestU01 tests. They may still be of interest in special situations, as they produce random numbers in a quite different fashion than all linear generators. If you doubt your simulation results, it may be an idea to redo a few simulations using an ICG or other nonlinear generator. The PRNG library by Otmar Lendl provides several such generators.[6]

19.3 Cryptographically Strong Random Number Generators

Cryptography, as well as gaming and gambling, have very different demands on random number generators than simulation-based science. For secure cryptography, it is crucial that a random number generator is *unpredictable*, i.e., that no efficient algorithm exists that permits an observer to predict future random numbers based on numbers produced by the generator in the past.

Reasonably fast cryptographic random number generators are based on encryption algorithms such as the Advanced Encryption Standard (AES) (Hellekalek and Wegenkittl 2003) and hash functions (Barker and Kelsey 2007). Encryption functions take a key K and a plaintext P and return a ciphertext

$$C = E(K, P). \tag{19.17}$$

They can be used as random number generators in one of three ways (Hellekalek and Wegenkittl 2003; Dworkin 2001):

- In *output feedback mode* (OFB) one chooses a fixed key K and some plaintext S to generate $C_0 = E(K, S)$. Random numbers are then generated by iterating the encryption on the ciphertext $C_j = E(K, C_{j-1})$.

[6] http://statistik.wu-wien.ac.at/software/prng.

- In *counter mode* (CTR) one chooses a fixed key K and obtains a sequence of ciphertexts by encrypting a counter that is increased by one for each number drawn, $C_j = E(K, S + j)$.
- *Key counter mode* (KTR) is complementary to counter mode: subsequent numbers are used as keys to encrypt a fixed plaintext, $C_j = E(S + j, P)$.

Random numbers are obtained from the ciphertext using a suitable drawing function $r_j = \mathcal{D}(C_j)$. The period of a generator based on (key) counter mode can obviously not be larger than the range of values the counter can assume. Therefore, long counters with hundreds of bits should be used. Several cryptographic RNGs have passed the TestU01 suite (L'Ecuyer and Simard 2007); the testsuite also contains implementations of random generators based on AES and SHA-1 encryption with periods of 2^{130} and 2^{440}, respectively.

Cryptographic RNGs in a (key) counter mode have no internal state variable: each random number is generated independently of all previous and subsequent numbers. This has two interesting advantages: First, skipping ahead is trivial. Second, many random numbers can be generated in parallel exploiting single-instruction, multiple-data (SIMD) coprocessors such as SSE and AltiVec extensions in modern CPUs or the massive power of current graphics cards (Tzeng and Wei 2008).

19.4 Seeding Random Number Generators

The choice of seeds for random number generators is subject of much folklore and little systematic investigation. Deficiencies in the seeding process of the Mersenne Twister and Knuth's LFG were discovered as late as 2002, and Matsumoto et al. (2007) recently reported that most PRNGs in the GNU Scientific Library have weak seeding strategies. Marsaglia and Tsang (2004) provide an interesting perspective on seeding of random number generators for legal or gambling purposes.

We assume here that surrogate data generation requires one to generate streams of random numbers to generate spike trains. All these streams must be statistically independent, no matter how they are generated and consumed: sequentially (serial algorithm runs many times in succession), simultaneously (many instances of a serial algorithm run independently on a cluster), or in parallel (parallel algorithm consuming several PRN streams).

Unfortunately no generator can provide *provably* statistically independent streams. The best a good PRNG can offer is to guarantee, or at least make it highly likely, that different seeds lead to nonoverlapping random number sequences of a certain length. A good PRNG should thus come with an explicit seeding algorithm which ensures reliable results for any admissible seed. Use it, instead of relying on traded advice on the choice of seeds such as choosing seeds with a roughly equal number of 0 and 1 bits.

The good news is that if the PRNG is well behaved for *any* choice of seed, then we can choose seeds systematically throughout our project. Assuming 32-bit num-

bers as seeds, we could, for example, compute the seed for any given stream according to

$$S_{\text{stream}} = n_{\text{coll}} \times 2^{26} + n_{\text{ex}} \times 2^{16} + n_{\text{tr}} \tag{19.18}$$

with collaborator number n_{coll}(1–63), experiment number n_{ex}(0–1,023), and trial number n_{tr}(0–65,535), specifying the stream. This scheme is easily extended to a wider range of components by using seeds with more than 32 bits. Such a scheme provides a systematic and clear way of enumerating random number streams, ensures that no two collaborators or simultaneous runs will use identical seeds, and leaves the difficult task of actually initializing the PRNG to the expert developers of generators.

That said, it is still useful to have a minimal understanding of how seeding actually works. For most PRNGs, the state vector $\mathbf{\Sigma}$ will take on any possible value at some point during the period ρ of the generator. Seeding simply initializes the state vector to some value, based on the value of the seed. Good seeding algorithms spread starting points evenly across the period ρ. When seeding the Mersenne Twister with $\rho = 2^{19,937} - 1$ with 32-bit integer seeds, starting points should be separated by roughly $2^{19,905}$ random numbers, provided that the seeding algorithm is good. Naive initialization of the state vector, on the other hand, may lead to initial states separated by only very few numbers and thus overlapping, highly correlated streams.

19.4.1 Parallel Streams of Random Numbers

We shall now briefly discuss the *block splitting* technique, also known as *cycle division* used to initialize PRNGs in a way that ensures nonoverlapping random number streams (L'Ecuyer et al. 2002; Bauke and Mertens 2007). *Parameterization* of PRNGs is another technique for the same purpose, see Mascagni and Srinivasan (2000, 2004).

Block splitting exploits that some PRNGs allow us to jump ahead by a certain number B of random numbers: given state $\mathbf{\Sigma}_0$, we can quickly find any $\mathbf{\Sigma}_{nB}$. Using n as a stream index, we can thus choose between ρ/B *provably* nonoverlapping random number streams of length B. The `MRG32k3a` generator (cf. Sect. 19.2.3) has a full period of $\rho \approx 2^{191}$, which can be split into 2^{64} streams of length $B = 2^{127}$ (L'Ecuyer et al. 2002). Similar techniques are available for lagged Fibonacci generators (Mascagni and Srinivasan 2004; Knuth 1998, Chap. 3.6, Ex. 3.6-9). Haramoto et al. (2008) developed a jump-ahead algorithm for the Mersenne Twister and related PRNGs, but no implementations appear to be publicly available at this time. Matlab v. 7.7 provides nonoverlapping streams of random numbers using block splitting for some generators; unfortunately, neither the Python `random.jumpahead()` (v. 2.6.2) nor the GSL `gsl_rng_set()` provide any similar guarantees.

Selecting random streams using block splitting does not mix well with changing the seed of the underlying PRNG, as both select starting points for random number

sequences from the same overall sequence. When using generators with multiple streams, you should choose one fixed seed for your entire study and then use (19.18) to select streams. For parallel simulations, simply extend the equation to include MPI rank or thread number.

19.5 Transforming Random Numbers

Most random number generators draw "raw" numbers r_j uniformly from the set $\mathbb{W} = \{0, 1, \ldots, M\}$ with $M \approx 2^{32}$. These raw numbers need to be transformed for use in simulations. We will consider here transformation to the unit interval, to a given range of integers, the exponential distribution, and some more general methods; for more information, see Gentle (2003), Knuth (1998), or Devroye (1986). If your software package or library provides methods to generate random deviates with the required distribution, you should use those methods.

Raw numbers r_j are usually converted to floating-point random numbers in the unit interval $[0, 1)$ using $u_j = r_j/(M + 1)$. While most scientific software today uses floating point variables with a 53-bit mantissa, yielding a machine resolution of $\mathcal{O}(10^{-16})$, 32-bit integer random numbers will yield u_j with a resolution of only $\mathcal{O}(10^{-9})$. This need not be a problem in practice, but you should be aware of this limitation. Random numbers with 53-bit mantissa can be obtained by suitable combination of two subsequent raw numbers.

The most common approach concerning the endpoints of the unit interval appears to be to draw from $[0, 1)$. Including 0 in the range of values can lead to problems in algorithms taking the logarithm of or dividing by random numbers. The most common strategies to exclude 0 are to redraw if 0 is drawn, or to substitute a very small value for 0. Doornik (2007) recently proposed an efficient way of obtaining floating point random numbers on the open interval $(0, 1)$ that are strictly symmetric about $\frac{1}{2}$. To our knowledge, his algorithm has not yet been incorporated in standard random number libraries.

Uniformly distributed integers i_j from a range $\{1, 2, \ldots, n\}$ are most easily computed as $i_j = 1 + \lfloor nu_j \rfloor$, although some libraries transform the r_j into i_j directly. You should not use $1 + (r_j \bmod n)$ to obtain the i_j, since this exploits only the lowest bits of the generated random numbers.

Random deviates following any other probability density $p(x)$ with pertaining cumulative distribution function $P(x) = \int_{-\infty}^{x} p(s)\,ds$ can be obtained in a number of ways. If $P(x)$ can be inverted analytically, then $x_j = P^{-1}(u_j)$ will be distributed according to $p(x)$. Exponentially distributed numbers with $p(x) = e^{-x/\mu}/\mu$ are thus obtained as $x_j = -\mu \ln(1 - u_j)$.

If $P^{-1}(x)$ is not available analytically or is computationally expensive, rejection methods are often useful. Let $q(x)$ be another probability density with an easily computable inverse $Q^{-1}(u)$, $p(x) \leq \alpha q(x)$ for all x, and $\alpha \geq 1$. Obtain a candidate $z = Q^{-1}(u_j)$; if $\alpha q(z)u_{j+1} < p(z)$, accept $x = z$, otherwise obtain a new candidate z. To be efficient, this method requires that the majorizing function $\alpha q(x)$ is a tight bound on the actual distribution $p(x)$. This can often be achieved by composing $q(x)$ from suitable functions for parts of the support of $p(x)$.

19.6 Recommendations

Donald Knuth (1998, Chap. 3.6) reminds us that

> "... the history of the subject warns us to be cautious. The most prudent policy ... to follow is to run each Monte Carlo program at least twice using quite different sources of random numbers, before taking the answers of the program seriously; this will not only give an indication of the stability of the results, it will also guard against the danger of trusting in a generator with hidden deficiencies. (Every random number generator will fail in at least one application.)"

Many a lesson has been learned the hard way about the interference between random number generators and Monte Carlo simulations in physics (Ferrenberg et al. 1992; Bauke and Mertens 2004). No comparable problems caused by random number generators have been reported in the computational neuroscience literature—yet. This does by no means mean that such problems do not exist: they might just have gone undetected so far, for want of analytical solutions to model equations providing gold standards. Spurious observations due to numerical inaccuracies in neuronal simulations should serve as a warning (Hansel et al. 1998).

With this in mind, we conclude with four recommendations:

1. Do not tie yourself to a single random number generator but implement a flexible interface to existing random number libraries.
2. Do not implement random number generators yourself but use existing, well-tested libraries such as the GNU Scientific Library.
3. Test your scheme for drawing random numbers against reference data provided by the RNG developers or against established test resources, such as L'Ecuyer's TestU01 testsuite, to make sure that random numbers are correctly handled by your interface to the random generator libraries. Include such tests in the test-suite for your software, so you can validate the interface when porting to a new architecture.
4. Consult *recent* tests of random number generators, at the time of writing the TestU01 test report (L'Ecuyer and Simard 2007).

And finally, before you submit your groundbreaking results for publication, repeat at least some simulation runs with at least one different random number generator.

References

Barker E, Kelsey J (2007) Recommendation for random number generation using deterministic random bit generators (revised). Technical Report NIST Special Publication 800-90. National Institute of Standards and Technology. http://csrc.nist.gov/publications/nistpubs/800-90/SP800-90revised_March2007.pdf

Bauke H, Mertens S (2004) Pseudo random coins show more heads than tails. J Stat Phys 114:1149–1169

Bauke H, Mertens S (2007) Random numbers for large-scale distributed Monte Carlo simulations. Phys Rev E (Statist Nonlin Soft Matter Phys) 75:066701. doi:10.1103/PhysRevE.75.066701. http://link.aps.org/abstract/PRE/v75/e066701

Brent RP (2006) Fast and reliable random number generators for scientific computing. In: Proceedings of the PARA'04 workshop on the state-of-the-art in scientific computing. Lect notes comput sci, vol 3732. Springer, Berlin, pp 1–10. http://wwwmaths.anu.edu.au/~brent/pub/pub217.html

Compagner A (1995) Operational conditions for random-number generation. Phys Rev E 52(5):5634–5645. doi:10.1103/PhysRevE.52.5634

Devroye L (1986) Non-uniform random variate generation. Springer, New York. Out of print. Available at http://cg.scs.carleton.ca/~luc/rnbookindex.html

Doornik JA (2007) Conversion of high-period random numbers to floating point. ACM Trans Model Comput Simul 17:3. http://doi.acm.org/10.1145/1189756.1189759

Dworkin M (2001) Recommendation for block cipher modes of operation: methods and techniques. Technical Report NIST Special Publication 800-38A. National Institute of Standards and Technology. http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf

Eichenauer J, Lehn J (1986) A nonlinear congruential pseudorandom number generator. Stat Hefte 27:315–326

Ferrenberg AM, Landau DP, Wong YJ (1992) Monte Carlo simulations: Hidden errors from "good" random number generators. Phys Rev Lett 69(23):3382–3384. doi:10.1103/PhysRevLett.69.3382

Galassi M, Davies J, Theiler J, Gough B, Jungman G, Booth M, Rossi F (2001) GNU scientific library reference manual. Network Theory, Bristol. http://sources.redhat.com/gsl

Gentle JE (2003) Random number generation and Monte Carlo methods, 2nd edn. Springer Science + Business Media, New York

Hansel D, Mato G, Meunier C, Neltner L (1998) On numerical simulations of integrate-and-fire neural networks. Neural Comput 10:467–483

Haramoto H, Matsumoto M, Nishimura T, Panneton F, L'Ecuyer P (2008) Efficient jump ahead for $\mathbb{F}_2$-linear random number generators. INFORMS J Comput 20(3):385–390. http://www.iro.umontreal.ca/~lecuyer/myftp/papers/jumpf2.pdf

Hellekalek P, Wegenkittl S (2003) Empirical evidence concerning AES. ACM Trans Model Comput Simul 13:322–333

Kim C, Choe GH, Kim DH (2008) Test of randomness by the gambler's ruin algorithm. Appl Math Comput 199:195–210. doi:10.1016/j.amc.2007.09.060

Knuth DE (1998) The art of computer programming, vol 2, 3rd edn. Addison-Wesley, Reading

L'Ecuyer P (1999) Good parameters and implementations for combined multiple recursive random number generators. Oper Res 47:159–164

L'Ecuyer P (2004) Random number generation. In: Gentle JE, Haerdle W, Mori Y (eds) Handbook of computational statistics. Springer, Berlin, pp 35–70. http://www.iro.umontreal.ca/~lecuyer/myftp/papers/handstat.pdf

L'Ecuyer P, Panneton F (2005) Fast random number generators based on linear recurrences modulo 2: overview and comparison. In: Kuhl ME, Steiger NM, Armstrong FB, Jones JA (eds) Proceedings of the 2005 winter simulation conference, pp 110–119

L'Ecuyer P, Simard R (2007) TestU01: A C library for empirical testing of random number generators. ACM Trans Math Softw 33:22. Article 22, 40 pages. doi:10.1145/1268776.1268777. http://www.iro.umontreal.ca/~simardr/testu01/tu01.html

L'Ecuyer P, Simard R, Chen EJ, Kelton WD (2002) An object-oriented random-number package with many long streams and substreams. Oper Res 50:1073–1075

Lewis PAW, Goodman AS, Miller JM (1969) A pseudo-random number generator for the System/360. IBM Syst J 8:136–146

Lüscher M (1994) A portable high-quality random number generator for lattice field theory simulations. Comput Phys Commun 79:100–110

Marsaglia G, Tsang WW (2004) The 64-bit universal RNG. Statist Probab Lett 66:183–187

Mascagni M, Srinivasan A (2000) Algorithm 806: SPRNG: a scalable library for pseudorandom number generation. ACM Trans Math Softw 26(3):436–461. http://doi.acm.org/10.1145/358407.358427

Mascagni M, Srinivasan A (2004) Parameterizing parallel multiplicative lagged-Fibonacci generators. Parallel Comput 30:899–916

Matsumoto M, Nishimura T (1998) Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator. ACM Trans Model Comput Simul 8:3–30

Matsumoto M, Saito M, Haramoto H, Nishimura T (2006) Pseudorandom number generation: impossibility and compromise. J Univers Comput Sci 12:672–690

Matsumoto M, Wada I, Kuramoto A, Ashihara H (2007) Common defects in initialization of pseudorandom number generators. ACM Trans Model Comput Simul 17:15. http://doi.acm.org/10.1145/1276927.1276928

Panneton F, L'Ecuyer P, Matsumoto M (2006) Improved long-period generators based on linear recurrences module 2. ACM Trans Math Softw 32:1–16

Park SK, Miller KW (1988) Random number generators: good ones are hard to find. Commun ACM 31:1192–1201

Plesser HE, Jahnsen AG (2010) Re-seeding invalidates tests of random number generators. Appl Math Comput 217:339–346. doi:10.1016/j.amc.2010.05.066

Press WH, Teukolsky SA, Vetterling WT, Flannery BP (1992) Numerical recipes in C, 2nd edn. Cambridge University Press, Cambridge

Saito M, Matsumoto M (2008) SIMD-oriented fast Mersenne twister: a 128-bit pseudorandom number generator. In: Keller A, Heinrich S, Niederreiter H (eds) Monte Carlo and quasi-Monte Carlo methods 2006. Springer, Berlin, pp 607–622

Tzeng S, Wei LY (2008) Parallel white noise generation on a GPU via cryptographic hash. In: I3D '08: Proceedings of the 2008 symposium on interactive 3D graphics and games. Microsoft Technical Report TR-2007-141. http://research.microsoft.com/research/pubs/view.aspx?tr_id=1384