CHAPTER 1

UNIFORM RANDOM NUMBER GENERATION

This chapter gives an overview of the main techniques and algorithms for generating uniform random numbers, including those based on linear recurrences, modulo 2 arithmetic, and combinations of these. A range of theoretical and empirical tests is provided to assess the quality of a uniform random number generator. We refer to Chapter 3 for a discussion on methods for random variable generation from arbitrary distributions — such methods are invariably based on uniform random number generators.

☞ 43

1.1 RANDOM NUMBERS

At the heart of any Monte Carlo method is a **random number generator**: a procedure that produces an infinite stream

$$U_1, U_2, U_3, \ldots \overset{\text{iid}}{\sim} \mathsf{Dist}$$

of random variables that are independent and identically distributed (iid) according to some probability distribution Dist. When this distribution is the uniform distribution on the interval (0,1) (that is, $\mathsf{Dist} = \mathsf{U}(0,1)$), the generator is said to be a **uniform random number generator**. Most computer languages already contain a built-in uniform random number generator. The user is typically requested only to input an initial number, called the **seed**, and upon invocation the random

number generator produces a sequence of independent uniform random variables on the interval $(0, 1)$. In MATLAB, for example, this is provided by the `rand` function.

The concept of an infinite iid sequence of random variables is a mathematical abstraction that may be impossible to implement on a computer. The best one can hope to achieve in practice is to produce a sequence of "random" numbers with statistical properties that are indistinguishable from those of a true sequence of iid random variables. Although physical generation methods based on universal background radiation or quantum mechanics seem to offer a stable source of such true randomness, the vast majority of current random number generators are based on simple algorithms that can be easily implemented on a computer. Following L'Ecuyer [10], such algorithms can be represented as a tuple $(\mathcal{S}, f, \mu, \mathcal{U}, g)$, where

- $\mathcal{S}$ is a finite set of **states**,
- f is a function from $\mathcal{S}$ to $\mathcal{S}$,
- μ is a probability distribution on $\mathcal{S}$,
- $\mathcal{U}$ is the **output space**; for a uniform random number generator $\mathcal{U}$ is the interval $(0, 1)$, *and we will assume so from now on, unless otherwise specified,*
- g is a function from $\mathcal{S}$ to $\mathcal{U}$.

A random number generator then has the following structure:

Algorithm 1.1 (Generic Random Number Generator)

1. **Initialize**: *Draw the seed S_0 from the distribution μ on $\mathcal{S}$. Set $t = 1$.*
2. **Transition**: *Set $S_t = f(S_{t-1})$.*
3. **Output**: *Set $U_t = g(S_t)$.*
4. **Repeat**: *Set $t = t + 1$ and return to Step 2.*

The algorithm produces a sequence $U_1, U_2, U_3, \ldots$ of **pseudorandom numbers** — we will refer to them simply as **random numbers**. Starting from a certain seed, the sequence of states (and hence of random numbers) must repeat itself, because the state space is finite. The smallest number of steps taken before entering a previously visited state is called the **period length** of the random number generator.

1.1.1 Properties of a Good Random Number Generator

What constitutes a good random number generator depends on many factors. It is always advisable to have a variety of random number generators available, as different applications may require different properties of the random generator. Below are some desirable, or indeed essential, properties of a good uniform random number generator; see also [39].

1. *Pass statistical tests*: The ultimate goal is that the generator should produce a stream of uniform random numbers that is indistinguishable from a genuine uniform iid sequence. Although from a theoretical point of view this criterion is too imprecise and even infeasible (see Remark 1.1.1), from a practical point

of view this means that the generator should pass a battery of simple statistical tests designed to detect deviations from uniformity and independence. We discuss such tests in Section 1.5.2.

2. *Theoretical support*: A good generator should be based on sound mathematical principles, allowing for a rigorous analysis of essential properties of the generator. Examples are linear congruential generators and multiple-recursive generators discussed in Sections 1.2.1 and 1.2.2.

3. *Reproducible*: An important property is that the stream of random numbers is reproducible without having to store the complete stream in memory. This is essential for testing and variance reduction techniques. Physical generation methods cannot be repeated unless the entire stream is recorded.

4. *Fast and efficient*: The generator should produce random numbers in a fast and efficient manner, and require little storage in computer memory. Many Monte Carlo techniques for optimization and estimation require billions or more random numbers. Current physical generation methods are no match for simple algorithmic generators in terms of speed.

5. *Large period*: The period of a random number generator should be extremely large — on the order of 10^{50} — in order to avoid problems with duplication and dependence. Evidence exists [36] that in order to produce N random numbers, the period length needs to be at least $10N^2$. Most early algorithmic random number generators were fundamentally inadequate in this respect.

6. *Multiple streams*: In many applications it is necessary to run multiple independent random streams in parallel. A good random number generator should have easy provisions for multiple independent streams.

7. *Cheap and easy*: A good random number generator should be cheap and not require expensive external equipment. In addition, it should be easy to install, implement, and run. In general such a random number generator is also more easily portable over different computer platforms and architectures.

8. *Not produce 0 or 1*: A desirable property of a random number generator is that both 0 and 1 are excluded from the sequence of random numbers. This is to avoid division by 0 or other numerical complications.

Remark 1.1.1 (Computational Complexity) From a theoretical point of view, a finite-state random number generator can *always* be distinguished from a true iid sequence, after observing the sequence longer than its period. However, from a practical point of view this may not be feasible within a "reasonable" amount of time. This idea can be formalized through the notion of *computational complexity*; see, for example, [33].

1.1.2 Choosing a Good Random Number Generator

As Pierre L'Ecuyer puts it [12], choosing a good random generator is like choosing a new car: for some people or applications speed is preferred, while for others robustness and reliability are more important. For Monte Carlo simulation the distributional properties of random generators are paramount, whereas in coding and cryptography unpredictability is crucial.

Nevertheless, as with cars, there are many poorly designed and outdated models available that should be avoided. Indeed several of the standard generators that come with popular programming languages and computing packages can be appallingly poor [13].

Two classes of generators that have overall good performance are:

1. *Combined multiple recursive generators*, some of which have excellent statistical properties, are simple, have large period, support multiple streams, and are relatively fast. A popular choice is L'Ecuyer's `MRG32k3a` (see Section 1.3), which has been implemented as one of the core generators in MATLAB (from version 7), VSL, SAS, and the simulation packages SSJ, Arena, and Automod.

2. *Twisted general feedback shift register generators*, some of which have very good equidistributional properties, are among the fastest generators available (due to their essentially binary implementation), and can have extremely long periods. A popular choice is Matsumoto and Nishimura's Mersenne twister `MT19937ar` (see Section 1.2.4), which is currently the default generator in MATLAB.

In general, a good uniform number generator has *overall* good performance, in terms of the criteria mentioned above, but is not usually the top performer over all these criteria. In choosing an appropriate generator it pays to remember the following.

- Faster generators are not necessarily better (indeed, often the contrary is true).
- A small period is in general bad, but a larger period is not necessarily better.
- Good equidistribution is a necessary requirement for a good generator but not a sufficient requirement.

1.2 GENERATORS BASED ON LINEAR RECURRENCES

The most common methods for generating pseudorandom sequences use simple linear recurrence relations.

1.2.1 Linear Congruential Generators

A **linear congruential generator** (LCG) is a random number generator of the form of Algorithm 1.1, with state $S_t = X_t \in \{0, \ldots, m-1\}$ for some strictly positive integer m called the **modulus**, and state transitions

$$X_t = (aX_{t-1} + c) \bmod m\,, \quad t = 1, 2, \ldots\,, \tag{1.1}$$

where the **multiplier** a and the **increment** c are integers. Applying the modulo-m operator in (1.1) means that $aX_{t-1} + c$ is divided by m, and the remainder is taken as the value for X_t. Note that the multiplier and increment may be chosen in the set $\{0, \ldots, m-1\}$. When $c = 0$, the generator is sometimes called a **multiplicative congruential generator**. Most existing implementations of LCGs are of this form

— in general the increment does not have a large impact on the quality of an LCG. The output function for an LCG is simply

$$U_t = \frac{X_t}{m} .$$

EXAMPLE 1.1 (Minimal Standard LCG)

An often-cited LCG is that of Lewis, Goodman, and Miller [24], who proposed the choice $a = 7^5 = 16807$, $c = 0$, and $m = 2^{31} - 1 = 2147483647$. This LCG passes many of the standard statistical tests and has been successfully used in many applications. For this reason it is sometimes viewed as the *minimal standard* LCG, against which other generators should be judged.

Although the generator has good properties, its period $(2^{31} - 2)$ and statistical properties no longer meet the requirements of modern Monte Carlo applications; see, for example, [20].

A comprehensive list of classical LCGs and their properties can be found on Karl Entacher's website:

`http://random.mat.sbg.ac.at/results/karl/server/`

The following recommendations for LCGs are reported in [20]:

- All LCGs with modulus 2^p for some integer p are badly behaved and should not be used.
- All LCGs with modulus up to $2^{61} \approx 2 \times 10^{18}$ fail several tests and should be avoided.

1.2.2 Multiple-Recursive Generators

A **multiple-recursive generator** (MRG) of **order** k is a random number generator of the form of Algorithm 1.1, with state $S_t = \mathbf{X}_t = (X_{t-k+1}, \ldots, X_t)^\top \in \{0, \ldots, m-1\}^k$ for some modulus m and state transitions defined by

$$X_t = (a_1 X_{t-1} + \cdots + a_k X_{t-k}) \bmod m , \quad t = k, k+1, \ldots , \tag{1.2}$$

where the **multipliers** $\{a_i, i = 1, \ldots, k\}$ lie in the set $\{0, \ldots, m-1\}$. The output function is often taken as

$$U_t = \frac{X_t}{m} .$$

The maximum period length for this generator is $m^k - 1$, which is obtained if (a) m is a prime number and (b) the polynomial $p(z) = z^k - \sum_{i=1}^{k-1} a_i z^{k-i}$ is *primitive* using modulo m arithmetic. Methods for testing primitivity can be found in [8, Pages 30 and 439]. To yield fast algorithms, all but a few of the $\{a_i\}$ should be 0.

MRGs with very large periods can be implemented efficiently by combining several smaller-period MRGs (see Section 1.3).

1.2.3 Matrix Congruential Generators

An MRG can be interpreted and implemented as a **matrix multiplicative congruential generator**, which is a random number generator of the form of Algorithm 1.1, with state $S_t = \mathbf{X}_t \in \{0, \ldots, m-1\}^k$ for some modulus m, and state transitions

$$\mathbf{X}_t = (A\mathbf{X}_{t-1}) \bmod m, \quad t = 1, 2, \ldots , \tag{1.3}$$

where A is an invertible $k \times k$ matrix and $\mathbf{X}_t$ is a $k \times 1$ vector. The output function is often taken as

$$\mathbf{U}_t = \frac{\mathbf{X}_t}{m} , \tag{1.4}$$

yielding a vector of uniform numbers in $(0,1)$. Hence, here the output space $\mathcal{U}$ for the algorithm is $(0,1)^k$. For fast random number generation, the matrix A should be sparse.

To see that the multiple-recursive generator is a special case, take

$$A = \begin{pmatrix} 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \\ a_k & a_{k-1} & \cdots & a_1 \end{pmatrix} \quad \text{and} \quad \mathbf{X}_t = \begin{pmatrix} X_t \\ X_{t+1} \\ \vdots \\ X_{t+k-1} \end{pmatrix} . \tag{1.5}$$

Obviously, the matrix multiplicative congruential generator is the k-dimensional generalization of the multiplicative congruential generator. A similar generalization of the multiplicative recursive generator — replacing the multipliers $\{a_i\}$ with matrices, and the scalars $\{X_t\}$ with vectors in (1.2) —, yields the class of **matrix multiplicative recursive generators**; see, for example, [34].

1.2.4 Modulo 2 Linear Generators

Good random generators must have very large state spaces. For an LCG this means that the modulus m must be a large integer. However, for multiple recursive and matrix generators it is not necessary to take a large modulus, as the state space can be as large as m^k. Because binary operations are in general faster than floating point operations (which are in turn faster than integer operations), it makes sense to consider random number generators that are based on linear recurrences modulo 2. A general framework for such random number generators is given in [18], where the state is a k-bit vector $\mathbf{X}_t = (X_{t,1}, \ldots, X_{t,k})^\top$ that is mapped via a linear transformation to a w-bit output vector $\mathbf{Y}_t = (Y_{t,1}, \ldots, Y_{t,w})^\top$, from which the random number $U_t \in (0,1)$ is obtained by *bitwise decimation* as follows. More precisely, the procedure is as follows.

Algorithm 1.2 (Generic Linear Recurrence Modulo 2 Generator)

1. **Initialize**: *Draw the seed* $\mathbf{X}_0$ *from the distribution* μ *on the state space* $\mathcal{S} = \{0,1\}^k$. *Set* $t = 1$.
2. **Transition**: *Set* $\mathbf{X}_t = A\mathbf{X}_{t-1}$.
3. **Output**: *Set* $\mathbf{Y}_t = B\mathbf{X}_t$ *and return*

$$U_t = \sum_{\ell=1}^{w} Y_{t,\ell}\, 2^{-\ell} .$$

Here, A and B are $k \times k$ and $w \times k$ binary matrices, respectively, and all operations are performed modulo 2. In algebraic language, the operations are performed over the finite field $\mathbb{F}_2$, where addition corresponds to the bitwise `XOR` operation (in particular, $1 + 1 = 0$). The integer w can be thought of as the word length of the computer (that is, $w = 32$ or 64). Usually (but there are exceptions, see [18]) k is taken much larger than w.

■ **EXAMPLE 1.2 (Linear Feedback Shift Register Generator)**

The **Tausworthe** or **linear feedback shift register** (LFSR) generator is an MRG of the form (1.2) with $m = 2$, but with output function

$$U_t = \sum_{\ell=1}^{w} X_{ts+\ell-1}\, 2^{-\ell} ,$$

for some $w \leqslant k$ and $s \geqslant 1$ (often one takes $s = w$). Thus, a binary sequence $X_0, X_1, \ldots$ is generated according to the recurrence (1.2), and the t-th "word" $(X_{ts}, \ldots, X_{ts+w-1})^\top$, $t = 0, 1, \ldots$ is interpreted as the binary representation of the t-th random number.

This generator can be put in the framework of Algorithm 1.2. Namely, the state at iteration t is given by the vector $\mathbf{X}_t = (X_{ts}, \ldots, X_{ts+k-1})^\top$, and the state is updated by advancing the recursion (1.2) over s time steps. As a result, the transition matrix A in Algorithm 1.2 is equal to the s-th power of the "1-step" transition matrix given in (1.5). The output vector $\mathbf{Y}_t$ is obtained by simply taking the first w bits of $\mathbf{X}_t$; hence $B = [I_w \;\; O_{w\times(k-w)}]$, where I_w is the identity matrix of dimension w and $O_{w\times(k-w)}$ the $w \times (k - w)$ matrix of zeros.

For fast generation most of the multipliers $\{a_i\}$ are 0; in many cases there is often only *one* other non-zero multiplier a_r apart from a_k, in which case

$$X_t = X_{t-r} \oplus X_{t-k} , \tag{1.6}$$

where $\oplus$ signifies addition modulo 2. The same recurrence holds for the states (vectors of bits); that is,

$$\mathbf{X}_t = \mathbf{X}_{t-r} \oplus \mathbf{X}_{t-k} ,$$

where addition is defined componentwise.

The LFSR algorithm derives its name from the fact that it can be implemented very efficiently on a computer via **feedback shift registers** — binary arrays that allow fast shifting of bits; see, for example, [18, Algorithm L] and [7, Page 40].

Generalizations of the LFSR generator that all fit the framework of Algorithm 1.2 include the **generalized feedback shift register** generators [25] and the **twisted** versions thereof [30], the most popular of which are the **Mersenne twisters** [31]. A particular instance of the Mersenne twister, `MT19937`, has become widespread, and has been implemented in software packages such as SPSS and MATLAB. It has a huge period length of $2^{19937} - 1$, is very fast, has good equidistributional properties, and passes most statistical tests. The latest version of the code may be found at

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

Two drawbacks are that the initialization procedure and indeed the implementation itself is not straightforward. Another potential problem is that the algorithm

recovers too slowly from the states near zero. More precisely, after a state with very few 1s is hit, it may take a long time (several hundred thousand steps) before getting back to some state with a more equal division between 0s and 1s. Some other weakness are discussed in [20, Page 23].

The development of good and fast modulo 2 generators is important, both from a practical and theoretical point of view, and is still an active area of research, not in the least because of the close connection to coding and cryptography. Some recent developments include the WELL (well-equidistributed long-period linear) generators by Panneton et al. [35], which correct some weaknesses in `MT19937`, and the SIMD-oriented fast Mersenne twister [38], which is significantly faster than the standard Mersenne twister, has better equidistribution properties, and recovers faster from states with many 0s.

1.3 COMBINED GENERATORS

A significant leap forward in the development of random number generators was made with the introduction of **combined generators**. Here the output of several generators, which individually may be of poor quality, is combined, for example by shuffling, adding, and/or selecting, to make a superior quality generator.

■ EXAMPLE 1.3 (Wichman–Hill)

One of the earliest combined generators is the Wichman–Hill generator [41], which combines three LCGs:

$$\begin{aligned} X_t &= (171\, X_{t-1}) \bmod m_1 \qquad (m_1 = 30269)\,, \\ Y_t &= (172\, Y_{t-1}) \bmod m_2 \qquad (m_2 = 30307)\,, \\ Z_t &= (170\, Z_{t-1}) \bmod m_3 \qquad (m_3 = 30323)\,. \end{aligned}$$

These random integers are then combined into a single random number

$$U_t = \frac{X_t}{m_1} + \frac{Y_t}{m_2} + \frac{Z_t}{m_3} \mod 1\,.$$

The period of the sequence of triples (X_t, Y_t, Z_t) is shown [42] to be $(m_1 - 1)(m_2 - 1)(m_3 - 1)/4 \approx 6.95 \times 10^{12}$, which is much larger than the individual periods. Zeisel [43] shows that the generator is in fact equivalent (produces the same output) as a multiplicative congruential generator with modulus $m = 27817185604309$ and multiplier $a = 16555425264690$.

The Wichman–Hill algorithm performs quite well in simple statistical tests, but since its period is not sufficiently large, it fails various of the more sophisticated tests, and is no longer suitable for high-performance Monte Carlo applications.

One class of combined generators that has been extensively studied is that of the **combined multiple-recursive generators**, where a small number of MRGs are combined. This class of generators can be analyzed theoretically in the same way as single MRG: under appropriate initialization the output stream of random numbers of a combined MRG is exactly the same as that of some larger-period

MRG [23]. Hence, to assess the quality of the generator one can employ the same well-understood theoretical analysis of MRGs. As a result, the multipliers and moduli in the combined MRG can be searched and chosen in a systematic and principled manner, leading to random number generators with excellent statistical properties. An important added bonus is that such algorithms lead to easy multi-stream generators [21].

In [12] L'Ecuyer conducts an extensive numerical search and detailed theoretical analysis to find good combined MRGs. One of the combined MRGs that stood out was `MRG32k3a`, which employs two MRGs of order 3,

$$X_t = (1403580\, X_{t-2} - 810728\, X_{t-3}) \bmod m_1 \quad (m_1 = 2^{32} - 209 = 4294967087)\,,$$

$$Y_t = (527612\, Y_{t-1} - 1370589\, Y_{t-3}) \bmod m_2 \quad (m_2 = 2^{32} - 22853 = 4294944443)\,,$$

and whose output is

$$U_t = \begin{cases} \dfrac{X_t - Y_t + m_1}{m_1 + 1} & \text{if } X_t \leqslant Y_t\,, \\[2ex] \dfrac{X_t - Y_t}{m_1 + 1} & \text{if } X_t > Y_t\,. \end{cases}$$

The period length is approximately 3×10^{57}. The generator `MRG32k3a` passes all statistical tests in today's most comprehensive test suit *TestU01* [20] (see also Section 1.5) and has been implemented in many software packages, including MATLAB, Mathematica, Intel's MKL Library, SAS, VSL, Arena, and Automod. It is also the core generator in L'Ecuyer's SSJ simulation package, and is easily extendable to generate multiple random streams. An implementation in MATLAB is given below.

```
%MRG32k3a.m
m1=2^32-209; m2=2^32-22853;
ax2p=1403580; ax3n=810728;
ay1p=527612; ay3n=1370589;

X=[12345 12345 12345]; % Initial X
Y=[12345 12345 12345]; % Initial Y

N=100; % Compute the sequence for N steps
U=zeros(1,N);
for t=1:N
   Xt=mod(ax2p*X(2)-ax3n*X(3),m1);
   Yt=mod(ay1p*Y(1)-ay3n*Y(3),m2);
   if Xt <= Yt
       U(t)=(Xt - Yt + m1)/(m1+1);
   else
       U(t)=(Xt - Yt)/(m1+1);
   end
   X(2:3)=X(1:2); X(1)=Xt; Y(2:3)=Y(1:2); Y(1)=Yt;
end
```

Different *types* of generators can also be combined. For example, Marsaglia's KISS99 (keep it simple stupid) generator [26] combines two shift register generators with an LCG. This generator performs very well in *TestU01* [20]. The following MATLAB code implements the KISS99 generator.

```
% KISS99.m
% Seeds: Correct variable types crucial!
A=uint32(12345); B=uint32(65435); Y=12345; Z=uint32(34221);
N=100; % Compute the sequence for N steps
U=zeros(1,N);
for t=1:N
    % Two Multiply with Carry Generators
    A=36969*bitand(A,uint32(65535))+bitshift(A,-16);
    B=18000*bitand(B,uint32(65535))+bitshift(B,-16);
    % MWC: Low and High 16 bits are A and B
    X=bitshift(A,16)+B;
    % CONG: Linear Congruential Generator
    Y = mod(69069*Y+1234567,4294967296);
    % SHR3: 3-Shift Register Generator
    Z=bitxor(Z,bitshift(Z,17));
    Z=bitxor(Z,bitshift(Z,-13));
    Z=bitxor(Z,bitshift(Z,5));
    % Combine them to form the KISS99 generator
    KISS=mod(double(bitxor(X,uint32(Y)))+double(Z),4294967296);
    U(t)=KISS/4294967296; % U[0,1] output
end
```

1.4 OTHER GENERATORS

Many variations on linear congruential methods have been proposed. Of the ones not discussed in the previous section we mention the following:

- *Multiply with carry:* This is a variation of the LCG where the increment c changes per iteration. Specifically, the recurrence is given by

$$X_t = (aX_{t-1} + c_{t-1}) \bmod m \,,$$

  where c_t (the **carry**) satisfies, for a given **lag** k,

$$c_t = \lfloor (aX_{t-k} + c_{t-1})/m \rfloor \,, \quad t \geqslant k \,.$$

- *XOR shift:* This is a generalization of an LFSR generator, and is a special case of a matrix MRG [34], where the state at iteration t is given by a binary vector $\mathbf{X}_t$ satisfying the linear recursion

$$\mathbf{X}_t = A_1 \mathbf{X}_{t-k_1} \oplus \cdots \oplus A_r \mathbf{X}_{t-k_r} \,,$$

  where $k_1, \ldots, k_r$ are strictly positive integers and $A_1, \ldots, A_r$ are either identity matrices or the products of XOR shift matrices.

- *Lagged Fibonacci generators:* This is a generalization of the LFSR generator (1.6), where the XOR operator $\oplus$ is replaced by a general binary operator, for example, the product.

More details on these generators can be found, for example, in [18, 29]. The above generators in general do not pass all statistical tests for randomness in the test suite *TestU01*, but combining them, as for example in the KISS99 generator, may produce high-quality generators. The multiply with carry and lagged Fibonacci generators are known to have poor theoretical properties [11, 40].

Congruential generators based on *nonlinear* recurrences,

$$X_t = g(X_{t-1}, \ldots, X_{t-k}) \bmod m \,,$$

for some nonlinear function g are currently not in much use in Monte Carlo simulations, since they tend to be slower, are more difficult to analyze theoretically, and often fail empirical tests for uniformity. However, nonlinear generators are important in cryptography, as the output sequence of linear congruential methods is easy to predict — in particular, the parameters of a linear congruential method can be easily estimated from previously generated output; see, for example, [29].

A famous nonlinear method in cryptography is that of Blum, Blum, and Shub [2], who proposed the quadratic recurrence

$$X_t = X_{t-1}^2 \bmod m \,,$$

where $m = p\,q$ and p and q are (large) primes that divided by 4 give a remainder of 3 (so-called **Blum primes**; for example, $p = 12676506002282294014967039815 19$ and $q = 12676506002282294014967043183 59$). Each iteration of the Blum–Blum–Shub generator produces only one bit of output, being either the even or odd bit parity, or the last bit (least significant bit) of X_t. It is shown in [2] that the output sequence of such a generator is not predictable in polynomial time. The generator is not appropriate for Monte Carlo simulation, due to its low speed.

Another example of a nonlinear congruential generator is the **inverse congruential generator** where the recurrence is of the form

$$X_t = (aX_{t-1}^{-} + c) \bmod m \,,$$

where X^- is the multiplicative inverse of X modulo m (that is, $XX^- = 1 \bmod m$ if it exists, or 0 otherwise). A survey of nonlinear generators may be found in [4].

1.5 TESTS FOR RANDOM NUMBER GENERATORS

The quality of random number generators can be assessed in two ways. The first is to investigate the theoretical properties of the random number generator. Such properties include the period length of the generator and various measures of uniformity and independence. This type of random number generator testing is called **theoretical**, as it does not require the actual output of the generator but only its algorithmic structure and parameters. Powerful theoretical tests are only feasible if the generators have a sufficiently simple structure, such as those of linear congruential and multiple-recursive methods and combined versions thereof.

A second type of test involves the application of a battery of statistical tests to the output of the generator, with the objective to detect deviations from uniformity and independence. Such tests are said to be **empirical**.

1.5.1 Spectral Test

One of the most useful theoretical tests concerns the structural properties of the generator. Suppose that $U_0, U_1, \ldots,$ is the sequence of numbers produced by a random number generator. It is well known [3, 5, 9, 27] that if the generator is of LCG or MRG type, then vectors of successive values $\mathbf{U}_0 = (U_0, \ldots, U_{d-1})^\top, \mathbf{U}_1 = (U_1, \ldots, U_d)^\top, \ldots,$ lie on a d-dimensional **lattice**; that is, a set $L \subset \mathbb{R}^d$ of the form

$$L = \left\{ \sum_{i=1}^{d} z_i \, \mathbf{b}_i, \quad z_1, \ldots, z_d \in \mathbb{Z} \right\},$$

for some set of linearly independent **basis** vectors $\mathbf{b}_1, \ldots, \mathbf{b}_d$. In other words, the elements of L are simply linear combinations of the basis vectors, using only integer coefficients. The lattice L is said to be **generated** by the basis matrix $B = (\mathbf{b}_1, \ldots, \mathbf{b}_d)$.

For an MRG satisfying the recursion (1.2), the basis vectors can be chosen as [15]

$$\begin{aligned}
\mathbf{b}_1 &= (1, 0, \ldots, 0, X_{1,k}, \ldots, X_{1,d-1})^\top / m \\
&\vdots \\
\mathbf{b}_k &= (0, 0, \ldots, 1, X_{k,k}, \ldots, X_{k,d-1})^\top / m \\
\mathbf{b}_{k+1} &= (0, 0, \ldots, 0, 1, \ldots, 0)^\top \\
&\vdots \\
\mathbf{b}_d &= (0, 0, \ldots, 0, 0, \ldots, 1)^\top,
\end{aligned}$$

where $X_{i,0}, X_{i,1}, \ldots$ is the sequence of states produced by the generator when starting with states $X_i = 1, X_t = 0, t \neq i, t \leqslant k$.

For a good generator the set $L \cap (0,1)^d$ should cover the d-dimensional unit hypercube $(0,1)^d$ in a uniform manner. One way to quantify this is to measure the distance between hyperplanes in the lattice L. The maximal distance between such hyperplanes is called the **spectral gap**, denoted here as g_d. A convenient way to compute the spectral gap is to consider first the **dual lattice** of L, which is the lattice generated by the inverse matrix of B. The dual lattice is denoted by L^*. Each vector $\mathbf{v}$ in L^* defines a family of equidistant hyperplanes in L, at a distance $1/\|\mathbf{v}\|$ apart. Hence, the length of the shortest non-zero vector in L^* corresponds to $1/g_d$.

For any d-dimensional lattice with m points there is a lower bound g_d^* on the spectral gap for dimension d. Specifically, for dimensions less than 8 it can be shown (see, for example, [8, Section 3.3.4]) that $g_d \geqslant g_d^* = \gamma_d^{-1/2} m^{-1/d}$, where $\gamma_1, \ldots, \gamma_8$ take the values

$$1, \quad (4/3)^{1/2}, \quad 2^{1/3}, \quad 2^{1/2}, \quad 2^{3/5}, \quad (64/3)^{1/6}, \quad 4^{3/7}, \quad 2\,.$$

An often-used *figure of merit* for the quality of a random number generator is the quotient

$$S_d = \frac{g_d^*}{g_d} = \frac{1}{g_d \, m^{1/d} \gamma_d^{1/2}},$$

or the minimum of K of such values: $S = \min_{d \leqslant K} S_d$, where $K \leqslant 8$. High values of S (close to 1) indicate that the generator has good structural properties.

The following example illustrates the main points; see also [8, Section 3.3.4].

EXAMPLE 1.4 (Lattice Structure and Spectral Gap)

Consider the LCG (1.1) with $a = 3$, $c = 0$, and $m = 31$. For $d = 2$, the corresponding lattice is generated by the basis matrix

$$B = \begin{pmatrix} 1/m & 0 \\ a/m & 1 \end{pmatrix},$$

since this LCG is an MRG with $k = 1$ and $X_{1,1} = a/m$. The dual lattice, which is depicted in Figure 1.1, is generated by the basis matrix

$$B^{-1} = \begin{pmatrix} m & 0 \\ -a & 1 \end{pmatrix}.$$

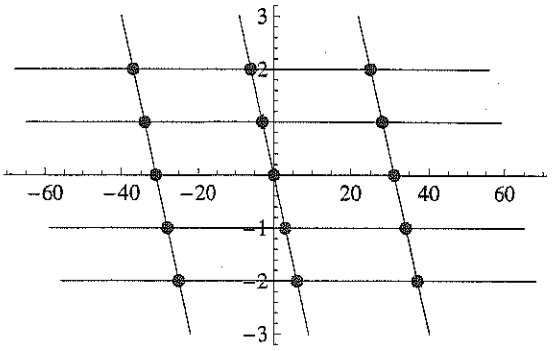

Figure 1.1 The dual lattice L^*.

The shortest non-zero vector in L^* is $(-3, 1)^\top$; hence, the spectral gap for dimension 2 is $g_2 = 1/\sqrt{10} \approx 0.316$. Figure 1.2 shows the normalized vector $g_2^2\,(-3, 1)^\top$ to be perpendicular to hyperplanes in L that are a distance g_2 apart. The figure of merit S_2 is here $3^{1/4}(5/31)^{1/2} \approx 0.53$.

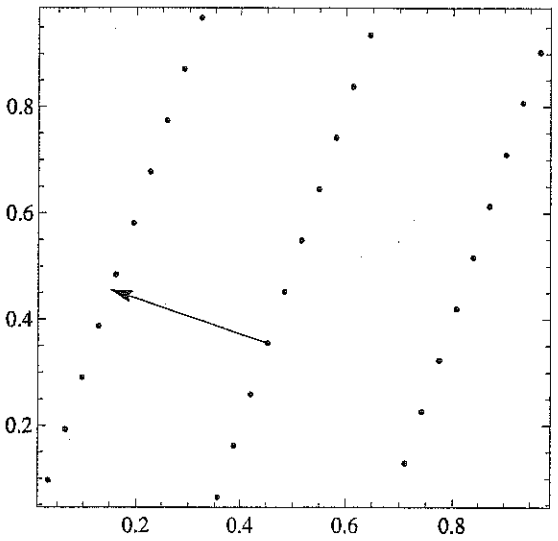

Figure 1.2 The lattice L truncated to the unit square. The length of the arrow corresponds to the spectral gap.

In order to select a good random number generator, it is important that the spectral gap is computed over a range of dimensions d. Some generators may display good structure at lower dimensions and bad structure at higher dimensions (the opposite is also possible). A classical example is IBM's RANDU LCG, with $a = 2^{16} + 3$, $c = 0$, and $m = 2^{31}$, which has reasonable structure for $d = 1$ and 2, but bad structure for $d = 3$; the latter is illustrated in Figure 1.3.

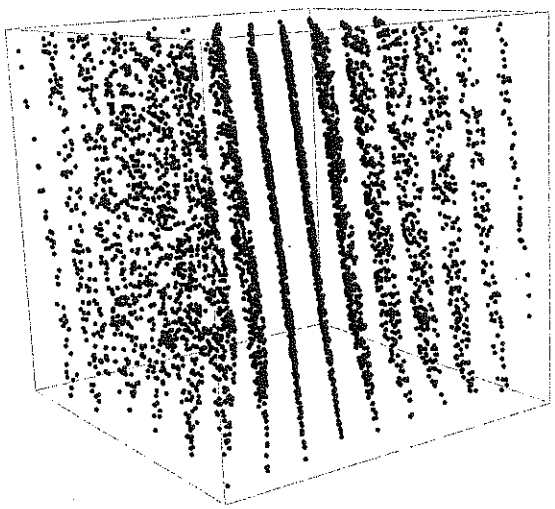

Figure 1.3 Structural deficiency of RANDU.

Structural properties of combined MRGs can be analyzed in the same way, as such generators are equivalent (under appropriate initialization conditions) to a single MRG with large modulus [23].

The computational effort required to compute the spectral gap grows rapidly with the dimension d and becomes impractical for dimensions over about 60. A fast implementation for analyzing the lattice structure of LCGs and MRGs is the LatMRG software package described in [17].

Modulo 2 linear generators do not have a lattice structure in Euclidean space, but they do in the space of formal power series. Much of the theory and algorithms developed for lattices in $\mathbb{R}^d$ carries over to the modulo 2 case [14].

Other theoretical tests of random number generators include discrepancy tests [32] and serial correlation tests [8, Section 3.3.3]. See also [1].

1.5.2 Empirical Tests

While theoretical tests are important for the elimination of bad generators and the search for potentially good generators [6, 12], the ultimate goal remains to find uniform random number generators whose output is statistically indistinguishable (within reasonable computational time) from a sequence of iid uniform random variables. Hence, any candidate generator should pass a wide range of statistical tests that examine uniformity and independence. The general structure of such tests is often of the following form.

Algorithm 1.3 (Two-Stage Empirical Test for Randomness) *Suppose that* $\mathbf{U} = \{U_i\}$ *represents the output stream of the uniform random generator. Let* H_0 *be the hypothesis that the* $\{U_i\}$ *are iid from a* $\mathsf{U}(0,1)$ *distribution. Let* Z *be some deterministic function of* $\mathbf{U}$.

1. *Generate* N *independent copies* $Z_1, \ldots, Z_N$ *of* Z *and evaluate a test statistic* $T = T(Z_1, \ldots, Z_N)$ *for testing* H_0 *versus the alternative that* H_0 *is not true. Suppose that under* H_0 *the test statistic* T *has distribution or asymptotic (for large* N*) distribution* Dist_0.

2. *Generate* K *independent copies* $T_1, \ldots, T_K$ *of* T *and perform a goodness of fit test to test the hypothesis that the* $\{T_i\}$ *are iid from* Dist_0.

Such a test procedure is called a **two-stage** or **second-order** statistical test. The first stage corresponds to an ordinary statistical test, such as a χ^2 goodness of fit test, and the second stage combines K such tests by means of another goodness of fit test, such as the Kolmogorov–Smirnov or Anderson–Darling test; see also Section 8.7.2. The following example demonstrates the procedure. ☞ 336

■ **EXAMPLE 1.5 (Binary Rank Test for the `drand48` Generator)**

The default random number generator in the C library is `drand48`, which implements an LCG with $a = 25214903917$, $m = 2^{48}$, and $c = 11$. We wish to examine if the output stream of this generator passes the *binary rank test* described in Section 1.5.2.11. For this test, the sequence $U_1, U_2, \ldots$ is first transformed to a binary sequence $B_1, B_2, \ldots$, for example, by taking $B_i = \mathrm{I}_{\{U_i \leqslant 1/2\}}$, and then the $\{B_i\}$ are arranged in a binary array, say with 32 rows and 32 columns. The first row of the matrix is $B_1, \ldots, B_{32}$, the second row is $B_{33}, \ldots B_{64}$, etc. Under H_0 the distribution of the rank (in modulo 2 arithmetic) R of this random matrix is given in (1.9). We generate $N = 200$ copies of R, and divide these into three classes: $R \leqslant 30$, $R = 31$, and $R = 32$. The expected number of ranks in these classes is by (1.9) equal to $E_1 = 200 \times 0.1336357$, $E_2 = 200 \times 0.5775762$, and $E_3 = 200 \times 0.2887881$. This is compared with the observed number of ranks O_1, O_2, and O_3, via the χ^2 goodness of fit statistic

$$T = \sum_{i=1}^{3} \frac{(O_i - E_i)^2}{E_i} \,. \tag{1.7}$$

Under H_0, the random variable T approximately has a χ^2_2 distribution (the number of degrees of freedom is the number of classes, 3, minus 1). This completes the first stage of the empirical test. ☞ 341

In the second stage, $K = 20$ replications of T are generated. The test statistics for the χ^2 test were 2.5556, 11.3314, 146.2747, 24.9729, 1.6850, 50.7449, 2.6507, 12.9015, 40.9470, 8.3449, 11.8191, 9.4470, 91.1219, 37.7246, 18.6256, 1.2965, 1.2267, 0.8346, 23.3909, 14.7596.

Notice that the null hypothesis would not be rejected if it were based only on the first outcome, 2.5556, as the p-value, $\mathbb{P}_{H_0}(T \geqslant 2.5556) \approx 0.279$ is quite large (and therefore the observed outcome is not uncommon under the null hypothesis). However, other values, such as 50.7449 are very large and lead to very small p-values (and a rejection of H_0). The second stage combines these findings into a single number, using a Kolmogorov–Smirnov test, to test whether the distribution

of T does indeed follow a χ^2_2 distribution. The empirical cdf (of the 20 values for T) and the cdf of the χ^2_2 distribution are depicted in Figure 1.4. The figure shows a clear disagreement between the two cdfs. The maximal gap between the cdfs is 0.6846 in this case, leading to a Kolmogorov–Smirnov test statistic value of $\sqrt{20} \times 0.6846 \approx 3.06$, which gives a p-value of around 3.7272×10^{-9}, giving overwhelming evidence that the output sequence of the `drand48` generator does not behave like an iid $\mathsf{U}(0,1)$ sequence.

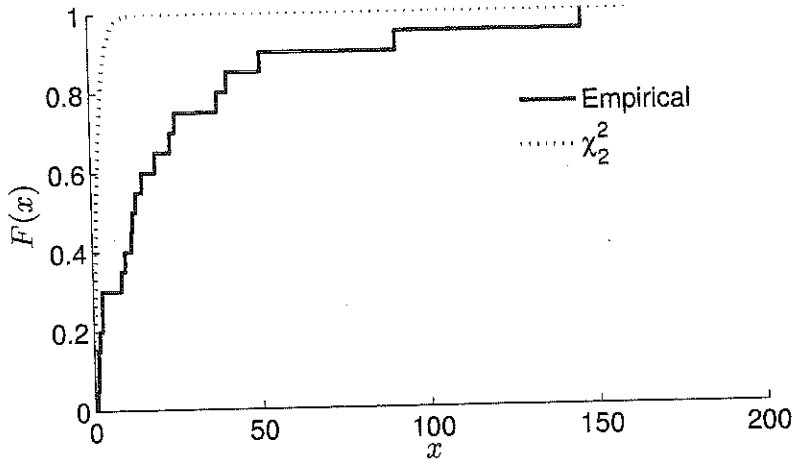

Figure 1.4 Kolmogorov–Smirnov test for the binary rank test using the `drand48` generator.

By comparison, we repeated the same procedure using the default MATLAB generator. The result of the Kolmogorov–Smirnov test is given in Figure 1.5. In this case the empirical and theoretical cdfs have a close match, and the p-value is large, indicating that the default MATLAB generator passes the binary rank test.

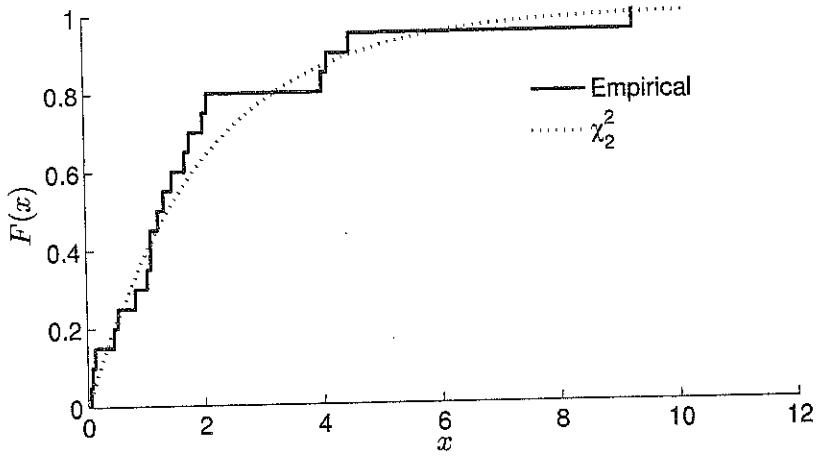

Figure 1.5 Kolmogorov–Smirnov test for the binary rank test using the default MATLAB random number generator (in this case the Mersenne twister).

Today's most complete library for the empirical testing of random number generators is the *TestU01* software library by L'Ecuyer and Simard [20]. The library is comprised of three predefined test suites: *Small Crush, Crush*, and *Big Crush*, in increasing order of complexity. *TestU01* includes the *standard tests* by Knuth [8, Section 3.3.2], and adapted version of the *Diehard* suite of tests by Marsaglia [28], the ones implemented by the National Institute of Standards and Technology (NIST) [37], and various other tests.

We conclude with a selection of empirical tests. Below, $U_0, U_1, \ldots$ is the original test sequence. The null hypothesis H_0 is that $\{U_i\} \sim_{\text{iid}} \mathsf{U}(0,1)$. Other random variables and processes derived from the $\{U_i\}$ are:

- $Y_0, Y_1, \ldots$, with $Y_i = \lfloor mU_i \rfloor$, $i = 0, 1, \ldots$, for some integer (*size*) $m \geqslant 1$. Under H_0 the $\{Y_i\}$ are iid with a discrete uniform distribution on $\{0, 1, \ldots, m-1\}$.
- $\mathbf{U}_0, \mathbf{U}_1, \ldots$, with $\mathbf{U}_i = (U_{id}, \ldots, U_{id+d-1})$, $i = 0, 1, \ldots$ for some dimension $d \geqslant 1$. Under H_0 the $\{\mathbf{U}_i\}$ are independent random vectors, each uniformly distributed on the d-dimensional hypercube $(0,1)^d$.
- $\mathbf{Y}_0, \mathbf{Y}_1, \ldots$, with $\mathbf{Y}_i = (Y_{id}, \ldots, Y_{id+d-1})$, $i = 0, 1, \ldots$ for some dimension $d \geqslant 1$. Under H_0 the $\{\mathbf{Y}_i\}$ are independent random vectors, each from the discrete uniform distribution on the d-dimensional set $\{0, 1, \ldots, m-1\}^d$.

1.5.2.1 Equidistribution (or Frequency) Tests This is to test whether the $\{U_i\}$ have a $\mathsf{U}(0,1)$ distribution. Two possible approaches are:

1. Apply a Kolmogorov–Smirnov test to ascertain whether the empirical cdf of $U_0, \ldots, U_{n-1}$ matches the theoretical cdf of the $\mathsf{U}(0,1)$ distribution; that is, $F(x) = x$, $0 \leqslant x \leqslant 1$.
2. Apply a χ^2 test on $Y_0, \ldots, Y_{n-1}$, comparing for each $k = 0, \ldots, m-1$ the observed number of occurrences in class k, $O_k = \sum_{i=0}^{n-1} \mathrm{I}_{\{Y_i = k\}}$, with the expected number $E_k = n/m$. Under H_0 the χ^2 statistic (1.7) asymptotically has (as $n \to \infty$) a χ^2_{m-1} distribution.

1.5.2.2 Serial Tests This is to test whether successive values of the random number generator are uniformly distributed. More precisely, generate vectors $\mathbf{Y}_0, \ldots, \mathbf{Y}_{n-1}$ for a given dimension d and size m. Count the number of times that the vector $\mathbf{Y}$ satisfies $\mathbf{Y} = \mathbf{y}$, for $\mathbf{y} \in \{0, \ldots, m-1\}^d$, and compare with the expected count n/m^d via a χ^2 goodness of fit test. It is usually recommended that each class should have enough samples, say at least 5 in expectation, so that $n \geqslant 5m^d$; however, see [22] for sparse serial tests. Typically, d is small, say 2 or 3.

1.5.2.3 Nearest Pairs Tests This is to detect spatial clustering (or repulsion) of the $\{\mathbf{U}_i\}$ vectors. Generate points (vectors) $\mathbf{U}_0, \ldots, \mathbf{U}_{n-1}$ in the d-dimensional unit hypercube $(0,1)^d$. For each pair of points $\mathbf{U}_i = (U_{i1}, \ldots, U_{id})^\top$ and $\mathbf{U}_j = (U_{j1}, \ldots, U_{jd})^\top$ let D_{ij} be the distance between them, defined by

$$D_{ij} = \begin{cases} \left[\sum_{k=1}^{d} (\min\{|U_{ik} - U_{jk}|, 1 - |U_{ik} - U_{jk}|\})^p\right]^{1/p} & \text{if} \quad 1 \leqslant p < \infty \\ \max_{k=1}^{d} \min\{|U_{ik} - U_{jk}|, 1 - |U_{ik} - U_{jk}|\} & \text{if} \quad p = \infty\,, \end{cases}$$

for some $1 \leqslant p \leqslant \infty$. This corresponds to the L^p norm on the *torus* $(0,1)^d$, whereby opposite sides of the unit hypercube are identified.

For $t \geqslant 0$, let N_t be the number of pairs (i, j), with $i < j$ such that $D_{ij} \leqslant (t/\lambda)^{1/d}$, where $\lambda = n(n-1)V_d/2$ and $V_d = [2\Gamma(1+1/p)]^d/\Gamma(1+d/p)$ (corresponding to the volume of the unit d-ball in L^p norm). It can be shown [16] that under H_0 the stochastic process $\{N_t, 0 \leqslant t \leqslant t_1\}$ converges in distribution (as $n \to \infty$) to a Poisson process with rate 1, for any fixed choice of t_1. It follows that if $T_1, T_2, \ldots$ are the jump times of $\{N_t\}$, then the spacings $A_i = T_i - T_{i-1}$, $i = 1, 2 \ldots$ are approximately iid Exp(1) distributed and the transformed spacings $Z_i = 1 - \exp(-A_i)$, $i = 1, 2 \ldots$ are approximately iid U(0, 1) distributed.

The **q-nearest pair** test assesses the hypothesis that the first q transformed spacings, $Z_1, \ldots, Z_q$, are iid from U(0, 1), by using a Kolmogorov–Smirnov or Anderson–Darling test statistic. By creating N copies of the test statistic, a two-stage test can be obtained.

Typically, ranges for the testing parameters are $1 \leqslant q \leqslant 8$, $1 \leqslant N \leqslant 30$, $2 \leqslant d \leqslant 8$, and $10^3 \leqslant n \leqslant 10^5$. The choice $p = \infty$ is often convenient in terms of computational speed. It is recommended [16] that $n \geqslant 4q^2\sqrt{N}$.

1.5.2.4 Gap Tests Let $T_1, T_2, \ldots$ denote the times when the output process $U_0, U_1, \ldots$, visits a specified interval $(\alpha, \beta) \subset (0, 1)$, and let $Z_1, Z_2, \ldots$ denote the **gap** lengths between subsequent visits; that is, $Z_i = T_i - T_{i-1} - 1$, $i = 1, 2, \ldots$, with $T_0 = 0$. Under H_0, the $\{Z_i\}$ are iid with a $\mathsf{Geom}_0(p)$ distribution, with $p = \beta - \alpha$; that is,

$$\mathbb{P}(Z = z) = p\,(1-p)^z, \quad z = 0, 1, 2, \ldots .$$

The gap test assesses this hypothesis by tallying the number of gaps that fall in certain classes. In particular, a χ^2 test is performed with classes $Z = 0, Z = 1, \ldots, Z = r-1$, and $Z \geqslant r$, with probabilities $p(1-p)^z$, $z = 0, \ldots, r-1$ for the first r classes and $(1-p)^r$ for the last class. The integers n and r should be chosen so that the expected number per class is $\geqslant 5$.

When $\alpha = 0$ and $\beta = 1/2$, this is sometimes called **runs above the mean**, and when $\alpha = 1/2$ and $\beta = 1$ this is sometimes called **runs below the mean**.

1.5.2.5 Poker or Partition Tests Consider the sequence of d-dimensional vectors $\mathbf{Y}_1, \ldots, \mathbf{Y}_n$, each taking values in $\{0, \ldots, m-1\}^d$. For such a vector $\mathbf{Y}$, let Z be the number of distinct components; for example if $\mathbf{Y} = (4, 2, 6, 4, 2, 5, 1, 4)$, then $Z = 5$. Under H_0, Z has probability distribution

$$\mathbb{P}(Z = z) = \frac{m(m-1)\cdots(m-z+1)\left\{{d \atop z}\right\}}{m^d}, \quad z = 1, \ldots, \min\{d, m\}\,. \tag{1.8}$$

Here, $\left\{{d \atop z}\right\}$ represents the **Stirling number of the second kind**, which gives the number of ways a set of size d can be partitioned into z non-empty subsets. For example, $\left\{{4 \atop 2}\right\} = 7$. Such Stirling numbers can be expressed in terms of binomial coefficients as

$$\left\{{d \atop z}\right\} = \frac{1}{z!}\sum_{k=0}^{z}(-1)^{z-k}\binom{z}{k}k^d\,.$$

Using the above probabilities, the validity of H_0 can now be tested via a χ^2 test.

1.5.2.6 Coupon Collector's Tests Consider the sequence $Y_1, Y_2, \ldots$, each Y_i taking values in $\{0, \ldots, m-1\}$. Let T be the first time that a "complete" set $\{0, \ldots, m-1\}$

is obtained among $Y_1, \ldots, Y_T$. The probability that $(Y_1, \ldots, Y_t)$ is incomplete is, by (1.8), equal to $\mathbb{P}(T > t) = 1 - m! \left\{ {t \atop m} \right\} / m^t$, so that

$$\mathbb{P}(T = t) = \frac{m!}{m^t} \left\{ {t-1 \atop m-1} \right\}, \quad t = m, m+1, \ldots .$$

The coupon collector's test proceeds by generating successive times $T_1, \ldots, T_n$ and applying a χ^2 goodness of fit test using classes $T = t$, $t = m, \ldots, r-1$ and $T > r-1$, with probabilities given above.

1.5.2.7 Permutation Tests Consider the d-dimensional random vector $\mathbf{U} = (U_1, \ldots, U_d)^\top$. Order the components from smallest to largest and let $\mathbf{\Pi}$ be the corresponding ordering of indices. Under H_0,

$$\mathbb{P}(\mathbf{\Pi} = \boldsymbol{\pi}) = \frac{1}{d!} \quad \text{for all permutations } \boldsymbol{\pi} .$$

The permutation test assesses this uniformity of the permutations via a χ^2 goodness of fit test with $d!$ permutation classes, each with class probability $1/d!$.

1.5.2.8 Run Tests Consider the sequence $U_1, U_2, \ldots$. Let Z be the **run-up length**; that is, $Z = \min\{k : U_{k+1} < U_k\}$. Under H_0, $\mathbb{P}(Z \geqslant z) = 1/z!$, so that

$$\mathbb{P}(Z = z) = \frac{1}{z!} - \frac{1}{(z+1)!}, \quad z = 1, 2, \ldots .$$

In the run test, n of such run lengths $Z_1, \ldots, Z_n$ are obtained, and a χ^2 test is performed on the counts, using the above probabilities. It is important to start from fresh after each run. In practice this is done by throwing away the number immediately after a run. For example the second run is started with U_{Z_1+2} rather than U_{Z_1+1}, since the latter is not $\mathsf{U}(0,1)$ distributed, as it is by definition smaller than U_{Z_1}.

1.5.2.9 Maximum-of-d Tests Generate $\mathbf{U}_1, \ldots, \mathbf{U}_n$ for some dimension d. For each $\mathbf{U} = (U_1, \ldots, U_d)^\top$ let $Z = \max\{U_1, \ldots, U_d\}$ be the maximum. Under H_0, Z has cdf

$$F(z) = \mathbb{P}(Z \leqslant z) = z^d, \quad 0 \leqslant z \leqslant 1 .$$

Apply the Kolmogorov–Smirnov test to $Z_1, \ldots, Z_n$ with distribution function $F(z)$. Another option is to define $W_k = Z_k^d$ and apply the equidistribution test to $W_1, \ldots, W_n$.

1.5.2.10 Collision Tests Consider a sequence of d-dimensional vectors $\mathbf{Y}_1, \ldots, \mathbf{Y}_b$, each taking values in $\{0, \ldots, m-1\}^d$. There are $r = m^d$ possible values for each $\mathbf{Y}$. Typically, $r \gg b$. Think of throwing b balls into r urns. As there are many more urns than balls, most balls will land in an empty urn, but sometimes a "collision" occurs. Let C be the number of such collisions. Under H_0 the probability of c collisions (that is, the probability that exactly $b - c$ urns are occupied) is given, as in (1.8), by

$$\mathbb{P}(C = c) = \frac{r(r-1)\cdots(r-(b-c)+1) \left\{ {b \atop b-c} \right\}}{r^b}, \quad c = 0, \ldots, b-1 .$$

A χ^2 goodness of fit test can be applied to compare the empirical distribution of n such collision values, $C_1, \ldots, C_n$, with the above distribution under H_0. One may need to group various of the classes $C = c$ in order to obtain a sufficient number of observations in each class.

1.5.2.11 Rank of Binary Matrix Tests Transform the sequence $U_1, U_2, \ldots$ to a binary sequence $B_1, B_2, \ldots$ and arrange these in a binary array of dimension $r \times c$ (assume $r \leqslant c$). Under H_0 the distribution of the rank (in modulo 2 arithmetic) Z of this matrix is given by

$$\mathbb{P}(Z = z) = 2^{(c-z)(z-r)} \prod_{i=0}^{z-1} \frac{(1-2^{i-c})(1-2^{i-r})}{1-2^{i-z}}, \quad z = 0, 1, \ldots, r\,. \tag{1.9}$$

☞ 632 This can be seen, for example, by defining a Markov chain $\{Z_t, t = 0, 1, 2, \ldots\}$, starting at 0 and with transition probabilities $p_{i,i} = 2^{-c+i}$ and $p_{i,i+1} = 1 - 2^{-c+i}$, $i = 0, \ldots, r$. The interpretation is that Z_t is the rank of a $t \times c$ matrix which is constructed from a $(t-1) \times c$ matrix by adding a $1 \times c$ random binary row; this row is either dependent on the $t-1$ previous rows (rank stays the same) or not (rank is increased by 1). The distribution of Z_r corresponds to (1.9).

For $c = r = 32$ we have

$$\begin{aligned}
\mathbb{P}(Z \leqslant 30) &\approx 0.1336357 \\
\mathbb{P}(Z = 31) &\approx 0.5775762 \\
\mathbb{P}(Z = 32) &\approx 0.2887881\,.
\end{aligned}$$

These probabilities can be compared with the observed frequencies, via a χ^2 goodness of fit test.

1.5.2.12 Birthday Spacings Tests Consider the sequence $Y_1, \ldots, Y_n$ taking values in $\{0, \ldots, m-1\}$. Sort the sequence as $Y_{(1)} \leqslant \ldots \leqslant Y_{(n)}$ and define spacings $S_1 = Y_{(2)} - Y_{(1)}, \ldots, S_{n-1} = Y_{(n)} - Y_{(n-1)}$, and $S_n = Y_{(1)} + m - Y_{(n)}$. Sort the spacings and denote them as $S_{(1)} \leqslant \ldots \leqslant S_{(n)}$.

Let R be the number of times that we have $S_{(j)} = S_{(j-1)}$ for $j = 1, \ldots, n$. The distribution of R depends on m and n, but for example when $m = 2^{25}$ and $n = 512$, we have [8, Page 71]:

$$\begin{aligned}
\mathbb{P}(R = 0) &\approx 0.368801577 \\
\mathbb{P}(R = 1) &\approx 0.369035243 \\
\mathbb{P}(R = 2) &\approx 0.183471182 \\
\mathbb{P}(R \geqslant 3) &\approx 0.078691997\,.
\end{aligned}$$

The idea is to repeat the test many times, say $N = 1000$, and perform a χ^2 test on the collected data. Asymptotically, for large n, R has a $\mathsf{Poi}(\lambda)$ distribution, with $\lambda = n^3/(4m)$, where λ should not be large; see [8, Page 570]. An alternative is to use $N = 1$ and base the decision whether to reject H_0 or not on the approximate p-value $\mathbb{P}(R \geqslant r) \approx 1 - \sum_{k=0}^{r-1} \mathrm{e}^{-\lambda} \lambda^k / k!$ (reject H_0 for small values). As a rule of thumb [19] the Poisson approximation is accurate when $m \geqslant (4N\lambda)^4$; that is, $Nn^3 \leqslant m^{5/4}$.

Further Reading

The problem of producing a collection of random numbers has been extensively studied, though as von Neumann said: "Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin." Nevertheless, we can produce numbers that are "sufficiently random" for much of the Monte Carlo simulation that occurs today. A comprehensive overview of random number generation can be found in [15]. The poor lattice structure of certain linear congruential generators was pointed out in [36], adding the concept of "good lattice structure" to the list of qualities a generator ought to have. Afflerbach [1] discusses a number of theoretical criteria for the assessment of random number generators. The celebrated Mersenne twister was introduced in [31], paving the way for generators with massive periods, which have become a necessity in the random number hungry world of Monte Carlo. A discussion of good multiple-recursive generators can be found in [12]. Niederreiter [33] covers many theoretical aspects of random number sequences, and Knuth [8] gives a classic treatment, discussing both the generation of random numbers and evaluation of the quality of same through the use of theoretical and empirical tests. The book by Tezuka [39] is exclusively on random numbers and proves a handy aid when implementing generators and tests. Books by Fishman [5] and Gentle [7] discuss the generation of random numbers for use in Monte Carlo applications. Our treatment of the spectral test draws from [5].

REFERENCES

1. L. Afflerbach. Criteria for the assessment of random number generators. *Journal of Computational and Applied Mathematics*, 31(1):3–10, 1990.
2. L. Blum, M. Blum, and M. Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing*, 15(2):364–383, 1986.
3. R. R. Coveyou and R. D. MacPherson. Fourier analysis of uniform random number generators. *Journal of the ACM*, 14(1):100–119, 1967.
4. J. Eichenauer-Herrmann. Pseudorandom number generation by nonlinear methods. *International Statistics Review*, 63(2):247–255, 1985.
5. G. S. Fishman. *Monte Carlo: Concepts, Algorithms and Applications.* Springer-Verlag, New York, 1996.
6. G. S. Fishman and L. R. Moore III. An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$. *SIAM Journal on Scientific and Statistical Computing*, 7(1):24–45, 1986.
7. J. E. Gentle. *Random Number Generation and Monte Carlo Methods.* Springer-Verlag, New York, second edition, 2003.
8. D. E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison-Wesley, Reading, MA, third edition, 1997.
9. P. L'Ecuyer. Random numbers for simulation. *Communications of the ACM*, 33(10):85–97, 1990.
10. P. L'Ecuyer. Uniform random number generation. *Annals of Operations Research*, 53(1):77–120, 1994.
11. P. L'Ecuyer. Bad lattice structure for vectors of non-successive values produced by linear recurrences. *INFORMS Journal of computing*, 9(1):57–60, 1997.

12. P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159 – 164, 1999.

13. P. L'Ecuyer. Software for uniform random number generation: distinguishing the good and the bad. In B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, editors, *Proceedings of the 2001 Winter Simulation Conference*, pages 95–105, Arlington, VA, December 2001.

14. P. L'Ecuyer. Polynomial integration lattices. In H. Niederreiter, editor, *Monte Carlo and Quasi-Monte Carlo methods*, pages 73–98, Berlin, 2002. Springer-Verlag.

15. P. L'Ecuyer. *Handbooks in Operations Research and Management Science: Simulation. S. G. Henderson and B. L. Nelson, eds.*, chapter 3: Random Number Generation. Elsevier, Amsterdam, 2006.

16. P. L'Ecuyer, J.-F. Cordeau, and R. Simard. Close-point spatial tests and their application to random number generators. *Operations Research*, 48(2):308–317, 2000.

17. P. L'Ecuyer and R. Couture. An implementation of the lattice and spectral tests for multiple recursive linear random number generators. *INFORMS Journal on Computing*, 9(2):206–217, 1997.

18. P. L'Ecuyer and F. Panneton. $\mathbb{F}_2$-linear random number generators. In C. Alexopoulos, D. Goldsman, and J. R. Wilson, editors, *Advancing the Frontiers of Simulation: A Festschrift in Honor of George Samuel Fishman*, pages 175–200, New York, 2009. Springer-Verlag.

19. P. L'Ecuyer and R. Simard. On the performance of birthday spacings tests with certain families of random number generators. *Mathematics and Computers in Simulation*, 55(1–3):131–137, 2001.

20. P. L'Ecuyer and R. Simard. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4), 2007. Article 22.

21. P. L'Ecuyer, R. Simard, E. J. Chen, and W. W. Kelton. An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50(6):1073–1075, 2002.

22. P. L'Ecuyer, R. Simard, and S. Wegenkittl. Sparese serial tests of uniformity for random number generators. *SIAM Journal of Scientific Computing*, 24(2):652–668, 2002.

23. P. L'Ecuyer and S. Tezuka. Structural properties for two classes of combined random number generators. *Mathematics of Computation*, 57(196):735–746, 1991.

24. P. A. Lewis, A. S. Goodman, and J. M. Miller. A pseudo-random number generator for the system/360. *IBM Systems Journal*, 8(2):136–146, 1969.

25. T. G. Lewis and W. H. Payne. Generalized feedback shift register pseudorandom number algorithm. *Journal of the ACM*, 20(3):456–468, 1973.

26. G. Marsaglia. KISS99. `http://groups.google.com/group/sci.stat.math/msg/b555f463a2959bb7/`.

27. G. Marsaglia. Random numbers fall mainly in the planes. *Proceedings of the National Academy of Sciences of the United States of America*, 61(1):25–28, 1968.

28. G. Marsaglia. DIEHARD: A battery of tests of randomness, 1996. `http://www.stat.fsu.edu/pub/diehard/`.

29. G. Marsaglia. Random number generators. *Journal of Modern Applied Statistical Methods*, 2(1):2–13, 2003.

30. M. Matsumoto and Y. Kurita. Twisted GFSR generators. *ACM Transactions on Modeling and Computer Simulation*, 2(3):179–194, 1992.

31. M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

32. H. Niederreiter. Recent trends in random number and random vector generation. *Annals of Operations Research*, 31(1):323–345, 1991.

33. H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods.* SIAM, Philadelphia, 1992.

34. H. Niederreiter. New developments in uniform pseudorandom number and vector generation. In H. Niederreiter and P. J.-S. Shiue, editors, *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, pages 87–120, New York, 1995. Springer-Verlag.

35. F. Panneton, P. L'Ecuyer, and M. Matsumoto. Improved long-period generators based on linear reccurences modulo 2. *ACM Transactions on Mathematical Software*, 32(1):1–16, 2006.

36. B. D. Ripley. The lattice structure of pseudo-random number generators. *Proceedings of the Royal Society, Series A*, 389(1796):197–204, 1983.

37. A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo. A statistical test suite for random and pseudorandom number generators for cryptographic applications. NIST special publication 800-22, National Institute of Standards and Technology, Gaithersburg, Maryland, USA, 2001. `http://csrc.nist.gov/rng/`.

38. M. Saito and M. Matsumoto. SIMD-oriented fast Mersenne twister: a 128-bit pseudorandom number generator. In *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pages 607 – 622, Berlin, 2008. Springer-Verlag.

39. S. Tezuka. *Uniform Random Numbers: Theory and Practice.* Springer-Verlag, New York, 1995.

40. S. Tezuka, P. L'Ecuyer, and R. Couture. On the add-with-carry and subtract-with-borrow random number generators. *ACM Transactions on Modeling and Computer Simulation*, 3(4):315–331, 1994.

41. B. A. Wichmann and I. D. Hill. Algorithm AS 183: An efficient and portable pseudo-random number generator. *Applied Statistics*, 31(2):188–190, 1982.

42. B. A. Wichmann and I. D. Hill. Correction to algorithm 183. *Applied Statistics*, 33(123), 1984.

43. H. Zeisel. Remark ASR 61: A remark on algorithm AS 183. an efficient and portable pseudo-random number generator. *Applied Statistics*, 35(1):89, 1986.