Contents

3

Uniform random numbers

Monte Carlo methods require a source of randomness. For the convenience of the users, random numbers are delivered as a stream of independent $\mathbf{U}(0,1)$ random variables. As we will see in later chapters, we can generate a vast assortment of random quantities starting with uniform random numbers. In practice, for reasons outlined below, it is usual to use simulated or ***pseudo-random numbers*** instead of genuinely random numbers.

This chapter looks at how to make good use of random number generators. That begins with selecting a good one. Sometimes it ends there too, but in other cases we need to learn how best to set seeds and organize multiple streams and substreams of random numbers. This chapter also covers quality criteria for random number generators, briefly discusses the most commonly used algorithms, and points out some numerical issues that can trap the unwary.

3.1 Random and pseudo-random numbers

It would be satisfying to generate random numbers from a process that according to well established understanding of physics is truly random. Then our mathematical model would match our computational method. Devices have been built for generating random numbers from physical processes such as radioactive particle emission, that are thought to be truly random. At present these are not as widely used as pseudo-random numbers.

A practical drawback to using truly random numbers is that they make it awkward to rerun a simulation after changing some parameters or discovering a bug. We can't restart the physical random number generator and so to rerun the simulation we have to store the sequence of numbers. Truly random sequences cannot be compressed, and so a lot of storage would be required. By contrast, a pseudo-random number generator only requires a little storage space for both code and internal data. When re-started in the same state, it re-delivers the same output.

A second drawback to physical random number generators is that they usually cannot supply random numbers nearly as fast as pseudo-random numbers can be generated. A third problem with physical numbers is that they may still fail some tests for randomness. This failure need not cast doubt on the underlying physics, or even on the tests. The usual interpretation is that the hardware that records and processes the random source introduces some flaws.

Because pseudo-random numbers dominate practice, we will usually drop the distinction, and simply refer to them as random numbers. No pseudo-random number generator perfectly simulates genuine randomness, so there is the possibility that any given application will resonate with some flaw in the generator to give a misleading Monte Carlo answer. When that is a concern, we can at least recompute the answers using two or more generators with quite different behavior. In fortunate situations, we can find a version of our problem that can be done exactly some other way to test the random number generator.

By now there are a number of very good and thoroughly tested generators. The best of these quickly produce very long streams of numbers, and have fast and portable implementations in many programming languages. Among these high quality generators, the Mersenne twister, MT19937, of Matsumoto and Nishimura (1998) has become the most prominent, though it is not the only high quality random number generator. We can not rule out getting a bad answer from a well tested random number generator, but we usually face much greater risks. Among these are numerical issues in which round-off errors accumulate, quantities overflow to ∞ or underflow to 0, as well as programming bugs and simulating from the wrong assumptions.

Sometimes a very bad random number generator will be embedded in general purpose software. The results of very extensive (and intensive) testing are reported in L'Ecuyer and Simard (2007). Many operating systems, programming languages and computing environments were found to have default random number generators which failed a great many tests of randomness. Any list of test results will eventually become out of date, as hopefully, the software it describes gets improved. But it seems like a safe bet that some bad random number generators will still be used as defaults for quite a while. So it is best to check the documentation of any given computing environment to be sure that good random numbers are used. The documentation should name the generator in use and give references to articles where theoretical properties and empirical tests have been published.

The most widely used random number generators for Monte Carlo sampling use simple recursions. It is often possible to observe a sequence of their outputs, infer their inner state and then predict future values. For some applications, such as cryptography it is necessary to have pseudo-random number sequences for which prediction is computationally infeasible. These are known as cryptographically secure random number generators. Monte Carlo sampling does not require cryptographic security.

3.2 States, periods, seeds, and streams

The typical random number generator provides a function, with a name such as `rand`, that can be invoked via an assignment like $x \leftarrow \texttt{rand}()$. The result is then a (simulated) draw from the $\mathbf{U}(0,1)$ distribution. The function `rand` might instead provide a whole vector of random numbers at once, but we can ignore that distinction in this discussion. Throughout this section we use pseudo-code with generic function names that resemble, without exactly matching, the names provided by real random number generators.

The function `rand` maintains a state vector. What goes on inside `rand` is essentially $\texttt{state} \leftarrow \texttt{update}(\texttt{state})$ followed by $\texttt{return}\ f(\texttt{state})$. After modifying the state, it returns some function of new state value. Here we will treat `rand` as a black box, and postpone looking inside until §3.4.

Even at this level of generality, one thing becomes clear. Because the state variable must have a finite size, the random number generator cannot go on forever without eventually revisiting a state it was in before. At that point it will start repeating values it already delivered.

Suppose that we repeatedly call $x_i \leftarrow \texttt{rand}()$ for $i \geqslant 1$ and that the state of the generator when x_{i_0} is produced is the same as it was when x_{i_0-P} was produced. Then $x_i = x_{i-P}$ holds for all $i \geqslant i_0$. From i_0 on, the generator is a deterministic cycle with period P. Although some generators cycle with different periods depending on what state they start in, we'll simplify things and suppose that the random number generator has a fixed period of P, so that $x_{i+P} = x_i$ holds for all i.

It is pretty clear that a small value of P makes for a poor simulation of random behavior. Random number generators with very large periods are preferred. One guideline is that we should not use more than about $\sqrt{P}$ random numbers from a given generator. Hellekalek and L'Ecuyer (1998) describe how an otherwise good linear congruential generator starts to fail tests when about $\sqrt{P}$ numbers are used. For the Mersenne twister, $P = 2^{19937} - 1 > 10^{6000}$, which is clearly ample. Some linear congruential generators discussed in §3.4 have $P = 2^{32} - 1$. These are far too small for Monte Carlo.

We can make a random number generator repeatable by intervening and setting its state before using it. Most random number generators supply a function like `setseed(seed)` that we can call. Here `seed` could be an encoding of the `state` variable. Or it may just be an integer that `setseed` translates into a full `state` variable for our convenience.

When we don't set the seed, some random number generators will seed themselves from a hidden source such as the computer's clock. Other generators use a prespecified default starting seed and will give the same random numbers every time they're used. That is better because it is reproducible. Using the system clock or some other unreproducible source damages not only the code itself but any other applications that call it. It is a good idea to set a value for the seed even if the random number generator does not require it. That way any interesting simulation values can be repeated if necessary, for example to track down a bug in the code that uses the random numbers.

By controlling the seed, we can synchronize multiple simulations. Suppose that we have J different simulations to do. They vary according to a problem specific parameter θ_j. One idiom for synchronization is:

```
given seed $s$, parameters $\theta_1, \dots, \theta_J$
for $j = 1, \dots, J$
  setseed($s$)
  $Y_j \leftarrow$ dosim($\theta_j$)
end for
return $Y_1, \dots, Y_J$
```

which makes sure that each of the J simulations start with the same random numbers. All the calls to `rand` take place within `dosim` or functions that it calls. Whether the simulations for different j remain synchronized depends on details within `dosim`.

Many random number generators provide a function called something like `getseed()` that we can use to retrieve the present state of the random number generator. We can follow `savedseed` $\leftarrow$ `getseed()` by `setseed(savedseed)` to restart a simulation where it left off. Some other simulation might take place in between these two calls.

In moderately complicated simulations, we want to have two or more streams of random numbers. Each stream should behave like a sequence of independent $\mathbf{U}(0,1)$ random variables. In addition, the streams need to appear as if they are statistically independent of each other. For example, we might use one stream of random numbers to simulate customer arrivals and another to simulate their service times.

We can get separate streams by carefully using `getseed()` and `setseed()`. For example in a queuing application, code like

```
setseed(arriveseed)
$A \leftarrow$ simarrive()
arriveseed $\leftarrow$ getseed()
```

alternating with

```
setseed(serviceseed)
$S \leftarrow$ simservice()
serviceseed $\leftarrow$ getseed()
```

gets us separated streams for these two tasks. Once again the calls to `rand()` are hidden, this time in `simarrive()` and `simservice()`.

Constant switching between streams is cumbersome and it is prone to errors. Furthermore it also requires a careful choice for the starting values of `arriveseed` and `serviceseed`. Poor choices of these seeds will lead to streams of random numbers that appear correlated with each other.

Algorithm 3.1 Substream usage

```
given $s$, n, nstep
setseed($s$)
for $i = 1, \dots, n$
  nextsubstream()
  $X_i \leftarrow$ oldsim(nstep)
  restartsubstream()
  $Y_i \leftarrow$ newsim(nstep)
end for
return $(X_1, Y_1), \dots, (X_n, Y_n)$
```

This algorithm shows pseudo-code for use of one substream per Monte Carlo replication, to run two simulation methods on the same random scenarios.

A better solution is to use a random number generator that supplies multiple streams of random numbers, and takes care of their seeds for us. With such a generator we can invoke

```
arriverng ← newrng()
servicerng ← newrng()
```

near the beginning of the simulation and then alternate $A \leftarrow$ `arriverng.rand()` with $S \leftarrow$ `servicerng.rand()` as needed. The system manages all the state vectors for us.

Some applications can make use of a truly enormous number of streams of random numbers. An obvious application is massively parallel computation, where every processor needs its own stream. Another use is to run a separate stream or substream for each of n Monte Carlo simulations. Algorithm 3.1 presents some pseudo-code for an example of this usage.

Algorithm 3.1 is for a setting with two versions of a simulation that we want to compare. Each simulation runs for `nstep` steps. Each simulated case i begins by advancing the random number stream to the next substream. After simulating `nstep` steps of the old simulation, it returns to the beginning of its random number substream, via `restartsubstream`, and then simulates the new method. Every new simulation starts off synchronized with the corresponding old simulation. Whether they stay synchronized depends on how their internal details are matched.

If we decide later to replace n by $n' > n$, then the first n results will be the same as we had before and will be followed by $n' - n$ new ones. By the same token, if we decide later to do longer simulations, replacing replacing `nstep` by a larger value `nstep`$'$, then the first `nstep` steps of simulation i will evolve as before. The longer simulations will retrace the initial steps of the shorter ones before extending them.

Comparatively few random number generators allow the user to have careful control of streams and substreams. A notable one that does is RngStreams by L'Ecuyer et al. (2002). Much of the description above is based on features of that software. They provide a sequence of about 2^{198} points in the unit interval. The generator is partitioned into streams of length 2^{127}. Each stream can be used as a random number generator. Every stream is split up into 2^{51} substreams of length 2^{76}. The design of RngStreams gives a large number of long substreams that behave nearly independently of each other. The user can set one seed to adjust all the streams and substreams.

For algorithms that consume random numbers on many different processors, we need to supply a seed to each processor. Each stream should still simulate independent $\mathbf{U}(0,1)$ random variables, but now we also want the streams to behave as if they were independent of each other. Making that work right depends on subtle details of the random number generator being used, and the best approach is to use a random number generator designed to supply multiple independent streams.

While design of random number generators is a mature field, design for parallel applications is still an active area. It is far from trivial to supply lots of good seeds to a random number generator. For users it would be convenient to be able to use consecutive non-negative integers to seed multiple generators. That may work if the generator has been designed with such seeds in mind, but otherwise it can fail badly.

Another approach is to use one random number generator to pick the seeds for another. That can fail too. Matsumoto et al. (2007) study this issue. It was brought to their attention by somebody simulating baseball games. The simulation for game i used a random number generator seeded by s_i. The seeds s_i were consecutive outputs of a second random number generator. These seeds resulted in the 15'th batter getting a hit in over 50% of the first 25 simulated teams, even though the simulation used batting averages near 0.250.

A better approach to getting multiple seeds for the Mersenne Twister is to take integers 1 to K, write them in binary, and apply a cryptographically secure hash function to their bits. The resulting hashed values are unlikely to show a predictable pattern. If they did, it would signal a flaw in the hash function. The advanced encryption standard (AES) (Daemen and Rijmen, 2002) provides one such hash function and there are many implementations of it.

3.3 $\mathbf{U}(0,1)$ random variables

The output of a random number generator is usually a simulated draw from the uniform distribution on the unit interval. There is more than one unit interval. Any of

$$[0,1], \quad (0,1), \quad [0,1) \quad \text{or} \quad (0,1]$$

can be considered to be the unit interval. But whichever interval we choose,

$$\mathbf{U}[0,1], \quad \mathbf{U}(0,1), \quad \mathbf{U}[0,1) \quad \text{and} \quad \mathbf{U}(0,1]$$

are all the same distribution. If $0 \leqslant a \leqslant b \leqslant 1$ and U is from any of these distributions then $\mathbb{P}(a \leqslant U \leqslant b) = b - a$. We could never distinguish between these distributions by sampling independently from them. If we ever got $U = 0$, then of course we would know it was from $\mathbf{U}[0,1)$ or $\mathbf{U}[0,1]$. But the probability of seeing $U = 0$ once in any finite sample is 0. Clearly, the same goes for seeing $U = 1$. Even with a countably infinite number of observations from $\mathbf{U}[0,1]$, we have 0 probability of seeing any exact 0s or 1s.

For some theoretical presentations it is best to work with the interval $[0,1)$. This half-open interval can be easily split into $k \geqslant 2$ congruent pieces $[(i-1)/k, i/k)$ for $i = 1,\dots,k$. Then we don't have to fuss with one differently shaped interval containing the value 1. By contrast, the open intervals $((i-1)/k, i/k)$ omit the values i/k, while the closed intervals $[(i-1)/k, i/k]$ duplicate some. Half-open intervals offer mainly a notational convenience.

In numerical work, real differences can arise between the various unit intervals. The random number generators we have considered simulate the uniform distribution on a discrete set of integers between 0 and some large value L. If the integer x is generated as the discrete uniform variable, then $U = x/L$ could be the delivered continuous uniform variable. It now starts to make a difference whether "between" 0 and L includes either 0, or L, or both. The result is a possibility that $U = 0$ or $U = 1$ can be delivered. Code that computes $-\log(1-U)$, or $1.0/U$ or $\sin(U)/U$ for example can then encounter problems.

It is best if the simulated values never deliver $U = 0$ or $U = 1$. That is something for the designer of the generator to arrange, perhaps by delivering $U = x/(L+1)$ in the example above. The user cannot always tell whether the random number generator avoids delivering 0s and 1s, partly because the random number generator might get changed later. As a result it pays to be cautious and write code that watches for $U = 0$ or 1. One approach is to reject such values and take the next number from the stream. But rejecting those values would destroy synchronization, with little warning to others using the code. Also, the function receiving the value U may not have access to the next element of the random number stream. As a result it seems better to interpret $U = 0$ as the smallest value of U that could be safely handled and similarly take $U = 1$ as the largest value that can be safely handled. In some circumstances it might be best to generate an error message upon seeing $U \in \{0,1\}$.

Getting $U = 0$ or $U = 1$ will be very rare if L is large, and the simulation is not. But for large simulations these values have a reasonable chance of being delivered. Suppose that $L \approx 2^{32}$ and the generator might deliver either a 0 or a 1. Then the chance of a problematic value is 2^{-31} and, when m numbers are drawn the expected number $m/2^{31}$ of problematic inputs could well be large.

The numerical unit interval differs from the mathematical one in another way. Floating point values are more finely spaced at the left end of the interval near 0 than they are at the right end, near 1. For example in double precision, we might find that there are no floating point numbers between 1 and $1 - 10^{-17}$ while the interval between 0 and 10^{-300} does have some. In single precision there is an even wider interval around 1 with no represented values. If uniform random numbers are used in single precision, then that alone could produce values of $U = 1.0$.

3.4 Inside a random number generator

Constructing random number generators is outside the scope of this book. Here we simply look at how they operate in order to better understand their behavior. The chapter end notes contain references which develop these results and many others.

The majority of modern random number generators are based on simple recursions using modular arithmetic. A well known example is the ***linear congruential generator*** or LCG:

$$x_i = a_0 + a_1 x_{i-1} \ \mathsf{mod}\ M. \tag{3.1}$$

Taking $a_0 = 0$ in the LCG, yields the ***multiplicative congruential generator*** or MCG:

$$x_i = a_1 x_{i-1} \ \mathsf{mod}\ M. \tag{3.2}$$

An LCG is necessarily slower than an MCG and because the LCG does not bring much if any quality improvement, MCGs are more widely used. A generalization of the MCG is the ***multiple recursive generator*** or MRG:

$$x_i = a_1 x_{i-1} + a_2 x_{i-2} + \cdots + a_k x_{i-k} \ \mathsf{mod}\ M, \tag{3.3}$$

where $k \geqslant 1$ and $a_k \neq 0$. ***Lagged Fibonacci generators*** which take the form $x_i = x_{i-r} + x_{i-s} \ \mathsf{mod}\ M$ for carefully chosen r, s and M are an important special case, because they are fast.

A relatively new and quite different generator type is the ***inversive congruential generator***, ICG. For a prime number M the ICG update is

$$x_i = \begin{cases} a_0 + a_1 x_{i-1}^{-1} \ \mathsf{mod}\ M & x_{i-1} \neq 0 \\ a_0 & x_{i-1} = 0. \end{cases} \tag{3.4}$$

When $x \neq 0$, then x^{-1} is the unique number in $\{0, 1, \ldots, M-1\}$ with $xx^{-1} = 1 \ \mathsf{mod}\ M$. The ICG behaves as if it uses the convention $0^{-1} = 0$.

These methods produce a sequence of integer values $x_i \in \{0, 1, \ldots, M-1\}$, that is, integers modulo M. With good choices for the constants a_j and M, the x_i can simulate independent random integers modulo M.

The LCG takes on at most M different values and so has period $P \leqslant M$. For the MCG we cannot allow $x_i = 0$ because then the sequence stays at 0. As a result $P \leqslant M - 1$ for the MCG. The MRG will start to repeat as soon as

k consecutive values $x_i, \dots, x_{i+k-1}$ duplicate some previously seen consecutive k-tuple. There are M^k of these, and just like with the MCG, the state with k consecutive 0s cannot be allowed. Therefore $P \leqslant M^k - 1$.

There are three main ways to choose M. Sometimes it is advantageous to choose M to be a large prime number. The value $2^{31} - 1$ is popular because it can be exploited in conjunction with 32 bit integers to get great speed. A prime number of the form $2^k - 1$ is called a ***Mersenne prime***. Another choice is to take $M = 2^r$ for some integer $r > 1$. For the MRG (3.3) it is reasonable to combine a small M with a large k to get a large value of $M^k - 1$. The third choice, with $M = 2$ is especially convenient because it allows fast bitwise operations to be used.

MRGs with $M = 2$ are called ***linear feedback shift register*** (LFSR) generators. They are also called ***Tausworthe generators***. With $M = 2$, the MRG delivers simulated random bits $x_i \in \{0, 1\}$. We might use 32 of them to produce an output value $u_i \in [0, 1]$. The original proposals took 32 consecutive bits to make u_i. Instead it is better to arrange for the lead bits of consecutive u_i to come from consecutive x's, as much as possible. Let the k'th bit of u_i be u_{ik}. In ***generalized feedback shift registers*** the bits u_{1k}, u_{2k}, $\dots$ are an LFSR for each k. A further operation, involving multiplication of those bits by a matrix (modulo 2), leads to a ***twisted GFSR*** (TGFSR). Twisting is a key ingredient of the Mersenne twister.

The MRG can be compactly written as,

$$\begin{pmatrix} x_i \\ x_{i-1} \\ x_{i-2} \\ \vdots \\ x_{i-k+1} \end{pmatrix} = \begin{pmatrix} a_1 & a_2 & \cdots & a_k \\ 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{pmatrix} \begin{pmatrix} x_{i-1} \\ x_{i-2} \\ x_{i-3} \\ \vdots \\ x_{i-k} \end{pmatrix} \mathsf{mod}\ M \tag{3.5}$$

or $\boldsymbol{s}_i = A\boldsymbol{s}_{i-1}\ \mathsf{mod}\ M$ for a state vector $\boldsymbol{s}_i = (x_i, x_{i-1}, \dots, x_{i-k+1})^\mathsf{T}$ and the given k by k matrix A with elements in $\{0, 1, \dots, M-1\}$. This matrix representation makes it easy to jump ahead in the stream of an MRG. To move ahead 2^ν places, we take

$$\boldsymbol{s}_{i+2^\nu} = (A^{2^\nu}\boldsymbol{s}_i)\ \mathsf{mod}\ M = ((A^{2^\nu}\ \mathsf{mod}\ M)\boldsymbol{s}_i)\ \mathsf{mod}\ M.$$

The matrix $A^{2^\nu}\ \mathsf{mod}\ M$ can be rapidly precomputed by the iteration $A^{2^\nu} = (A^{2^{\nu-1}})^2\ \mathsf{mod}\ M$.

Equation (3.5) also makes it simple to produce and study thinned out substreams $\widetilde{x}_i \equiv x_{\ell+ki}$, based on taking every k'th output value. Those substreams are MRGs with $\widetilde{A} = A^k\ \mathsf{mod}\ M$. The thinned out stream is not necessarily better than the original one. In the case of an MCG, the thinned out sequence is also an MCG with $\widetilde{a}_1 = a_1^k\ \mathsf{mod}\ M$. If this value were clearly better, then we would probably be using it instead of a_1.

The representation (3.5) includes the MCG as well, for $k = 1$, and similar, though more complicated formulas, can be obtained for the LCG after taking account of the constant a_0.

a	a^2	a^3	a^4	a^5	a^6
0	0	0	0	0	0
1	1	1	1	1	1
2	4	1	2	4	1
3	2	6	4	5	1
4	2	1	4	2	1
5	4	6	2	3	1
6	1	6	1	6	1

Table 3.1: The first column shows values of a ranging from 0 to 6. Subsequent columns show powers of a taken modulo 7. The last 6 entries of the final column are all 1's in accordance with Fermat's theorem. The primitive roots are 3 and 5.

3.5 Uniformity measures

For simplicity, suppose that our random number generator produces values x_i for $i \geqslant 1$ and that it has a single cycle with period P, so that $x_{i+P} = x_i$. The first thing to ask of a generator is that the values $x_1, \dots, x_P$ when converted to $u_i \in [0,1]$ should be approximately uniformly distributed. Though P is usually too large for us to in fact produce a histogram of $u_1, \dots, u_P$, the mathematical structure in the generator allows comparison of that histogram to the $\mathbf{U}(0,1)$ distribution.

Naturally, P genuinely independent $\mathbf{U}(0,1)$ random variables would not have a perfectly flat histogram. We prefer a perfectly flat histogram for $u_1, \dots, u_P$ because an uneven one would over-sample some part of the unit interval and whichever part it oversampled would bring an unreasonable bias. For P large enough, the genuinely random values would have a very flat histogram with high probability.

Designers of random number generators carefully choose parameters for us. Suppose that the MCG has $x_{n+1} \equiv a^n x_1 \mathsf{~mod~} m$ where $m = p$ is a prime number. If $a^n \equiv 1 \mathsf{~mod~} p$ then $x_{n+k} = x_k$ for all k, and the period is at most n. For each a let $r(a) = \min\{k > 0 \mid a^k \equiv 1 \mathsf{~mod~} p\}$. We know that the period cannot be any larger than $p-1$ because there are only $p-1$ nonzero integers modulo p. A primitive root is a value a with $r(a) = p-1$, attaining the maximum period. Table 3.1 illustrates that for $p = 7$ there are two primitive roots $a = 3$ and $a = 5$. For a large p, the designer of a random number generator may restrict attention to primitive roots, but then has to consider other criteria to choose among those roots.

Simply getting good uniformity in the values $u_1, \dots, u_M$ is not enough. The LCG

$$x_n = x_{n-1} + 1 \mathsf{~mod~} m \tag{3.6}$$

achieves perfect uniformity on values 0 through $m-1$ by going through them in order, and so it produces a perfect discrete uniform distribution for u_i. Of course, it badly violates the independence we require for successive pairs of random numbers.

The way that bad generators like (3.6) are eliminated is by insisting that the pairs $(u_1,u_2),(u_2,u_3),\dots,(u_P,u_1)$ have approximately the $\mathbf{U}[0,1]^2$ distribution. The RANDU number generator Lewis et al. (1969), was designed in part to get the best distribution for Spearman correlations of 10 consecutive points (u_i,u_{i+L}) at lags $L\in\{1,2,3\}$. RANDU was a very popular generator. Unfortunately, RANDU has very bad uniformity. The consecutive triples (u_i,u_{i+1},u_{i+2}) are all contained within 15 parallel planes in $[0,1]^3$. RANDU is now famous for its unacceptable three dimensional distribution, and not for the care taken in tuning the two dimensional distributions.

Ideally the k-tuples $(u_i,u_{i+1},\dots,u_{i+k-1})$ for general k should also be very uniform, at least for the smaller values of k. The pseudo-random number generators that we use can only gaurantee uniformity up to some moderately large values of k.

If a good random number generator is to have near uniformity in the values of $(u_i,u_{i+1},\dots,u_{i+k-1})\in[0,1]^k$, then the search for good generators requires a means of measuring the degree of uniformity in a list of k-tuples.

A very direct assessment of uniformity begins by splitting the unit interval $[0,1)$ into 2^ℓ congruent subintervals $[a/2^\ell,(a+1)/2^\ell)$ for $0\leqslant a<2^\ell$. The subinterval containing u_i can be determined from the first ℓ bits in u_i. The unit cube $[0,1)^k$ is similarly split into $2^{k\ell}$ subcubes. A random number generator with period $P=2^K$ is ***k*-distributed to *ℓ*-bit accuracy** if each of the boxes

$$B_{\boldsymbol{a}}\equiv\prod_{j=1}^{k}\Bigl[\frac{a_j}{2^\ell},\frac{a_j+1}{2^\ell}\Bigr),$$

for $0\leqslant a_j<2^\ell$, has $2^{K-k\ell}$ of the points $(u_i,u_{i+1},\dots,u_{i+k-1})$ for $i=1,\dots,P$. We also say that such a random number generator is **(k,ℓ)-equidistributed**. We will see a more powerful form of equidistribution, using not necessarily cubical regions, in Chapter 15 on quasi-Monte Carlo.

Many random number generators have a period of the form $P=2^K-1$, because the state vector is not allowed to be all zeros. Such random number generators are said to be (k,ℓ)-equidistributed if each of the boxes $B_{\boldsymbol{a}}$ above has $2^{K-k\ell}$ points in it, except the one for $\boldsymbol{a}=(0,0,\dots,0)$, which then has $2^{K-k\ell}-1$ points in it.

The Mersenne twister MT19937 is 623-distributed to 32 bits accuracy. It is also 1246-distributed to 16 bits accuracy, 2492-distributed to 8 bits accuracy, 4984-distributed to 4 bits accuracy, 9968-distributed to 2 bits accuracy and 19937-distributed to 1 bit accuracy. One reason for its popularity is that the dimension $k=623$ for which equidistribution is proven is relatively large. This fact may be more important than the enormous period which MT19937 has.

Marsaglia (1968) showed that the consecutive tuples $(u_i,\dots,u_{i+k-1})$ from an MCG have a lattice structure. Figure 3.1 shows some examples for $k=2$, using p small enough for us to see the structure.

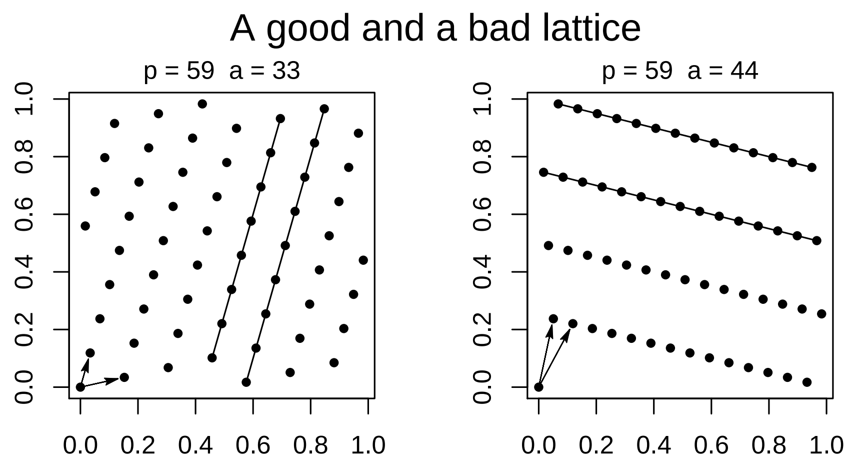

Figure 3.1: This figure illustrates two very small MCGs. The plotted points are (u_i, u_{i+1}) for $1 \leqslant i \leqslant P$. The lattice structure is clearly visible. Both MCGs have $M = 59$. The multiplier a_1 is 33 in the left panel and 44 in the right. Two basis vectors are given near the origin. The points lie in systems of parallel lines as shown.

A lattice is an infinite set of the form

$$\mathcal{L} = \Bigl\{ \sum_{j=1}^{k} \alpha_j v_j \mid \alpha_j \in \mathbb{Z} \Bigr\}, \tag{3.7}$$

where v_j are linearly independent basis vectors in $\mathbb{R}^k$. The tuples from the MCG are the intersection of the infinite set $\mathcal{L}$ with the unit cube $[0,1)^k$. The definition of a lattice closely resembles the definition of a vector space. The difference is that the coefficients α_j are integers in a lattice, whereas they are real values in a vector space. We will study lattices further in Chapters 15 through 17 on quasi-Monte Carlo. There is more than one way to choose the vectors $v_1, \dots, v_k$. Each panel in Figure 3.1 shows one basis (v_1, v_2) by arrows from the origin.

The lattice points in k dimensions lie within sets of parallel $k-1$ dimensional planes. Lattices where those planes are far apart miss much of the space, and so are poor approximations to the desired uniform distribution. In the left panel of Figure 3.1, the two marked lines are distance 0.137 apart. The points could also be placed inside a system of lines 0.116 apart, nearly orthogonal to the set shown. The first number is the quality measure for these points because we seek to minimize the maximum separation. The lattice on the right is worse, because the maximum separation, shown by two marked lines, is 0.243.

The number of planes on which the points lie tends to be smaller when those planes are farther apart. The relationship is not perfectly monotone because the number of parallel planes required to span the unit cube depends on both their separation and on the angles that they make with the coordinate axes of the cube. For an MCG with period P, Marsaglia (1968) shows that there is always a system of $(k!P)^{1/k}$ or fewer parallel planes that contain all of the k-tuples.

MCGs are designed in part to optimize some measure of quality for the planes in which their k-tuples lie. Just as a bad lattice structure is evidence of a flaw in the random number generator, a good lattice structure, combined with a large value of P is proof that the k dimensional projections of the full period are very uniform. Let $d_k = d_k(u_1, \dots, u_P)$ be the maximum separation between planes in $[0,1]^k$ for the k-tuples formed from $u_1, \dots, u_P$. We want a small value for d_1 and d_2, and remembering RANDU, for d_k at larger values of k. It is customary to use ratios $s_k = d_k^*/d_k$ where d_k^* is some lower bound for d_k given the period P. The idea is to seek a generator with large (near 1) values for s_k for all small k. Computing d_k can be quite difficult and the choice of a generator involves trade-offs between the quality at multiple values of k. Gentle (2003, Chapter 2.2) has a good introduction to tests of lattice structure.

Inversive congruential generators produce points that do not have a lattice structure. Nor do they satisfy strong (k,ℓ)-equidistribution properties. Their uniformity properties are established by showing that the fraction of their points within any box $\prod_{j=1}^{k}[0,a_j]$ is close to the volume $\prod_j a_k$ of that box. We will revisit this and other measures of uniformity, called discrepancies, in Chapter 15 on quasi-Monte Carlo.

3.6 Statistical tests of random numbers

The (k,ℓ)-equidistribution and spectral properties mentioned in §3.5 apply to the full period of the random number generator. They describe uniformity for the set of all P k-tuples of the form $(x_i, x_{i+1}, \dots, x_{i+k-1})$ for $1 \leqslant i \leqslant P$ and small enough k. In applications, we use only a small number $n \ll P$ of the random numbers. We might then be concerned that individual small portions of a generator could behave badly even though they work well when taken together.

To address this concern, some tests have been designed for small subsets of random number generator output. Some of the tests are designed to mimic the way random number generators might be used in practice. Others are designed to probe for specific flaws that we suspect or know appear in some random number generators.

Dozens of these tests are now in use. A few examples should give the idea. The references in the chapter end notes describe the tests in detail.

One test described by Knuth (1998) is based on permutations. Take k consecutive output values u_i, u_{i+1}, …, u_{i+k-1} from the sequence where $k \geqslant 1$ is a small integer, such as 5. If the output values are ranked from smallest to largest, then there are $k!$ possible orderings. All $k!$ orderings should be equally probable. We can test uniformity by finding n non-overlapping k-tuples, counting the number of times each of $k!$ orderings arises and running Pearson's χ^2 test to compare the observed counts versus the expected values $n/k!$.

The χ^2 tests gives a p-value

$$p = \mathbb{P}\Bigl(\chi^2_{(k!-1)} \geqslant \sum_{\ell=1}^{k!} \frac{(O_\ell - E_\ell)^2}{E_\ell}\Bigr)$$

where O_ℓ is the number of times that ordering ℓ was observed and $E_\ell = n/k!$ is the expected number of times, and $\chi^2_{(\nu)}$ is a chi-squared random variable on ν degrees of freedom. Small values of p are evidence that the u_i are not IID uniform. If the u_i are IID $\mathbf{U}(0,1)$, then the distribution of p approaches the $\mathbf{U}(0,1)$ distribution as $n \to \infty$.

A simple test of uniformity is to take large n and then compute a p-value such as the one given above. If p is very small then we have evidence of a problem with the random number generator. It is usually not enough to find and report whether one p-value from a given test is small or otherwise. Instead we want small, medium and large p-values to appear with the correct relative frequency for sample sizes n comparable to ones we might use in practice. In a ***second-level test*** we repeat a test like the one above some large number N of times and obtain p-values $p_1, \dots, p_N$. These should be nearly uniformly distributed. The second level test statistic is a measure of how close $p_1, \dots, p_N$ are to the $\mathbf{U}(0,1)$ distribution. A popular choice is the Kolmogorov-Smirnov statistic

$$\mathrm{KS} = \sup_{0 \leqslant t \leqslant 1} \Bigl| \frac{1}{N} \sum_{j=1}^{N} \mathbb{1}_{p_j \leqslant t} - t \Bigr|.$$

The second level p-value is $\mathbb{P}(Q_N \geqslant \mathrm{KS})$ where Q_N is a random variable that truly has the distribution of a Kolmogorov-Smirnov statistic applied to N IID $\mathbf{U}(0,1)$ random variables. That is a known distribution, which makes the second level test possible. Small values, such as 10^{-10} for the second order p-value indicate a problem with the random number generator. Alternatives to Kolmogorov-Smirnov, such as the Anderson-Darling test, have more power to capture departures from $\mathbf{U}(0,1)$ in the tails of the distribution of p_j.

A second-level test will detect random number generators that are too uniform, having for example too few p_j below 0.01. They will also capture subtle flaws like p values that are neither too high nor too low on average, but instead tend to be close to 0.5 too often.

There are third-level tests based on the distribution of second-level test statistics but the most practically relevant tests are at the first or second level. We finish this section by describing a few more first level tests which can be used to construct corresponding second order tests.

Marsaglia's ***birthday test*** has parameters m and n. In it, the RNG is used to simulate m birthdays $b_1, \dots, b_m$ in a hypothetical year of n days. Each b_i should be $\mathbf{U}\{1, 2, \dots, n\}$. Marsaglia and Tsang (2002) consider $n = 2^{32}$ and $m = 2^{12}$. The sampled b_i values are sorted into $b_{(1)} \leqslant b_{(2)} \leqslant \cdots \leqslant b_{(m)}$ and differenced, forming $d_i = b_{(i+1)} - b_{(i)}$ for $i = 1, \dots, m-1$. The test statistic is D, the number of values among the spacings d_i that appear two or more times. The distribution of D is roughly $\mathrm{Poi}(\lambda)$ where $\lambda = m^3/(4n)$ for m and n in the range they use. For the given m and n, we find $\lambda = 4$. The test compares N sample values of D to the Poisson distribution. The birthday test seems strange, but it is known to detect problems with some lagged Fibonacci generators.

A good multiplicative congruential generator has its k-tuples on a well separated lattice. Those points are then farther apart from each other than P random points in $[0,1)^k$ would be. Perhaps a small sample of n points $\boldsymbol{v}_j = (u_{jk+1}, \dots, u_{(j+1)k})$ for $j = 1, \dots, n$ preserves this problem and the points avoid each other too much. The ***close-pair*** tests are based on the distribution of $\mathrm{MD}_{n,k} = \min_{1 \leqslant j < j' \leqslant n} \|\boldsymbol{v}_j - \boldsymbol{v}_{j'}\|$. Here $\|\boldsymbol{z}\|$ is a convenenient norm. L'Ecuyer et al. (2000) find that norms with a wraparound feature, (treating 0 and 1 as identical) are convenient because boundary effects disappear making it easier to approximate the distribution that $\mathrm{MD}_{n,k}$ would have given perfect uniform numbers.

To run these tests we need to know the true distribution of their test statistics on IID $\mathbf{U}(0,1)$ inputs. That true distribution is usually easiest to work out if the k-tuples in the test statistic are non-overlapping $(u_{jk+1}, \dots, u_{(j+1)k})$ for $j = 1, \dots, n$. Sometimes the tests are run instead on overlapping k-tuples $(u_{j+1}, \dots, u_{j+k})$ for $j = 1, \dots, n$. Tests on overlapping tuples are perfectly valid, though it may then be much harder to find the proper distribution of the test statistic.

3.7 Pairwise independent random numbers

Suppose that we really want to use physical random numbers, and that we stored nd simulated $\mathbf{U}(0,1)$ random variables from a source that has passed enough empirical testing to win our trust. We can put these together to obtain n points $\boldsymbol{U}_1, \dots, \boldsymbol{U}_n \sim \mathbf{U}(0,1)^d$. We may find that n is too small. Then it is possible, as described below, to construct $N = n(n-1)/2$ pairwise independent variables $\boldsymbol{X}_1, \dots, \boldsymbol{X}_N$ from $\boldsymbol{U}_1, \dots, \boldsymbol{U}_n$. This much larger number of random variables may be enough for our Monte Carlo simulation.

The construction makes use of addition modulo 1. For $z \in \mathbb{R}$, define $\lfloor z \rfloor$ to be the ***integer part*** of z, namely the largest integer less than or equal to z. Then $z \bmod 1$ is $z - \lfloor z \rfloor$. The sum of u and v modulo 1 is $u + v \bmod 1 = u + v - \lfloor u + v \rfloor$. We write $u + v \bmod 1$ as $u \oplus v$ for short. The N vectors $\boldsymbol{X}_i$ are obtained as $\boldsymbol{U}_j \oplus \boldsymbol{U}_k$ for all pairs $1 \leqslant j < k \leqslant n$. Addition modulo 1 is applied separately for each component of $\boldsymbol{X}_i$.

A convenient iteration for generating the $\boldsymbol{X}_i$ is

```
$i \leftarrow 0$
**for** $j = 1, \dots, n-1$
**for** $k = j+1, \dots, n$
   $i \leftarrow i + 1$
   $\boldsymbol{X}_i \leftarrow \boldsymbol{U}_j \oplus \boldsymbol{U}_k$
**end double for loop**
```
(3.8)

In a simulation we ordinarily use $\boldsymbol{X}_i$ right after it is created.

The vectors $\boldsymbol{X}_i$ are ***pairwise independent*** (Exercise 3.9). This means that $\boldsymbol{X}_i$ is independent of $\boldsymbol{X}_{i'}$ for $1 \leqslant i < i' \leqslant N$. They are not fully independent. For example

$$(\boldsymbol{U}_1 \oplus \boldsymbol{U}_2) + (\boldsymbol{U}_3 \oplus \boldsymbol{U}_4) - (\boldsymbol{U}_1 \oplus \boldsymbol{U}_3) - (\boldsymbol{U}_2 \oplus \boldsymbol{U}_4)$$

is always an integer. As a result, the $\boldsymbol{X}_i$ can be used in problems where pairwise independence, but not full independence, is enough.

Proposition 3.1. *Let $Y = f(\boldsymbol{X})$ have mean μ and variance $\sigma^2 < \infty$ when $\boldsymbol{X} \sim \mathbf{U}(0,1)^d$. Suppose that $\boldsymbol{X}_1, \dots, \boldsymbol{X}_N$ are pairwise independent $\mathbf{U}(0,1)^d$ random variables. Let $Y_i = f(\boldsymbol{X}_i)$, $\bar{Y} = (1/N)\sum_{i=1}^N Y_i$ and $s^2 = (N-1)^{-1}\sum_{i=1}^N (Y_i - \bar{Y})^2$. Then*

$$\mathbb{E}(\bar{Y}) = \mu, \quad \mathrm{Var}(\bar{Y}) = \frac{\sigma^2}{N}, \quad \text{and} \quad \mathbb{E}(s^2) = \sigma^2.$$

Proof. Exercise 3.10. □

Pairwise independence differs from full independence in one crucial way. The average $\bar{Y}$ of pairwise independent and identically distributed random variables does not necessarily satisfy a central limit theorem. The distribution depends on how the pairwise independent points are constructed.

To get a 99% confidence interval for $\mathbb{E}(Y)$ we could form R genuinely independent replicates $\bar{Y}_1, \dots, \bar{Y}_R$, each of which combines N pairwise independent random variables, and then use the interval

$$\bar{\bar{Y}} \pm 2.58\biggl(\frac{1}{R(R-1)}\sum_{r=1}^R (\bar{Y}_r - \bar{\bar{Y}})^2\biggr)^{1/2}, \quad \text{where} \quad \bar{\bar{Y}} = \frac{1}{R}\sum_{r=1}^R \bar{Y}_r.$$

A central limit applies as R increases, assuming that $0 < \sigma^2 < \infty$.

Chapter end notes

Extensive background on random number generators appears in Gentle (2003), Tezuka (1995) and Knuth (1998). The latter book includes a description of philosophical issues related to using deterministic sequences to simulate random ones and a thorough treatment of congruential generators and spectral tests. Gentle (2003, Chapter 1) describes and relates many different kinds of random number generators. L'Ecuyer (1998) provides a comprehensive survey of random number generation.

Multiplicative congruential generators were first proposed by Lehmer (1951). Linear feedback shift register generators are due to Tausworthe (1965). Together, these are the antecedents of most of the modern random number generators.

GFSRs were introduced by Lewis and Payne (1973) and TGFSRs by Matsumoto and Kurita (1992). The Mersenne twister was introduced in Matsumoto and Nishimura (1998).

TestU01 is a comprehensive test package for random number generators is given by L'Ecuyer and Simard (2007). It incorporates the tests from Marsaglia's diehard test battery, available on the Internet, as well as tests developed by the National Institute of Standards and Technology. When a battery of tests is applied to an ensemble of random number generators, we not only see which generators fail some test, we also see which tests fail many generators. The latter pattern can even help spot tests that have errors in them (Leopardi, 2009).

For cautionary notes on parallel random number generation see Hellekalek (1998), Mascagni and Srinivasan (2004) and Matsumoto et al. (2007). The latter describe several parametrization techniques for picking seeds. They recommend a seeding scheme from L'Ecuyer et al. (2002).

Owen (2009) investigates the use of pairwise independent random variables drawn from physical random numbers, described in §3.7. The average

$$\bar{Y} = \frac{1}{\binom{n}{2}} \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} f(\boldsymbol{U}_j \oplus \boldsymbol{U}_k) \tag{3.9}$$

has a limiting distribution, as $n \to \infty$ that is symmetric (weighted sum of differences of χ^2) but not Gaussian. Because it is non-Gaussian, some independent replications of $\bar{Y}$ are needed as described in §3.7 in order to apply a central limit theorem. Because it is symmetric, we do not need a large number of replications.

There are many proposals to make new random number generators from old ones. Such hybrid, or ***combined random number generators*** can be used to get a much longer period, potentially as long as the product of the component random number generators' periods, or to eliminate flaws from one generator. For example, the lattice structure of an MCG can be broken up by suitable combinations with inversive generators (L'Ecuyer and Granger-Piché, 2003). Similarly, for applications in which the output must not be predictable, a hybrid with physical random numbers may be suitable.

One way to combine two random number streams u_i and v_i is to deliver $u_i \oplus v_i$ where $\oplus$ is addition modulo 1 as described in §3.7. A second class of techniques is to use the output of one random number generator to shuffle the output of another. Knuth (1998) remarks that shuffling makes it hard to skip ahead in the random number generator. Combined generators can be difficult to analyze. L'Ecuyer (1994, Section 9) reports that some combinations make matters worse while others bring improvements. Therefore a combined generator should be selected using the same care that we use to pick an ordinary generator.

The update matrix A in (3.5) is sparse. When we want to jump ahead by 2^μ places, then we use A^{2^ν} which may not be sparse. In the case of the Mersenne Twister, large powers of A yield a dense 19937×19937 matrix of bits, taking over 47 megabytes of memory and requiring slow multiplication. RngStreams avoids this problem because it is a hybrid of several smaller random number generators, and jumping ahead requires a modest number of small dense matrices. Improved methods of jumping ahead in the Mersenne Twister have been developed (Haramoto et al., 2008) but they do not recommend how far to jump ahead in order to form streams.

Recommendations

For Monte Carlo applications, it is necessary to use a high quality random number generator with a long enough period. The theoretical principles underlying the generator and its quality should be published. There should also be a published record showing how well the generator does on a standard battery of tests, such as TestU01.

When using an RNG, it is good practice to explicitly set the seed. That allows the computations to be reproduced later.

If many independent streams are required then a random number generator which supports them is preferable. For example RngStreams produces independent streams and has been extensively tested.

In moderately large projects, there is an advantage to isolating the random number generator inside one module. That makes it easier to replace the random numbers later if there are concerns about their quality.

Exercises

3.1. Let $P = 2^{19937} - 1$ be the period of the Mersenne twister. Using the equidistribution properties of the Mersenne twister:

a) For how many $i = 1, \dots, P$ will we find $\max_{1 \leqslant j \leqslant 623} u_{i+j-1} < 2^{-32}$?

b) For how many $i = 1, \dots, P$ will we find $\min_{1 \leqslant j \leqslant 623} u_{i+j-1} \geqslant 1 - 2^{-32}$?

c) Supose that we use the Mersenne twister to simulate coin tosses, with toss i being heads if $u_i < 1/2$ and tails if $1/2 \leqslant u_i$. Is there any index i in $1 \leqslant i \leqslant P$ for which $u_i, \dots, u_{i+10000-1}$ would give 10,000 heads in a row? How about 20,000 heads in a row?

Note that when $i + j - 1 > P$ the value u_{i+j-1} is still well defined. It is $u_{i+j-1 \bmod P}$.

3.2. The lattice on the right of Figure 3.1 is not the worst one for $p = 59$. Find another value of a for which the period of $x_i = ax_{i-1} \bmod 59$, starting with $x_1 = 1$ equals 59, but the 59 points (u_i, u_{i+1}) for $u_i = x_i/59$ lie on parallel lines more widely separated than those with $a = 44$. Plot the points and compute the separation between those lines. [Hint: identify a lattice point on one of the lines, and drop a perpendicular from it to a line defined by two points on another of the lines.]

3.3. Suppose that we are using an MCG with $P \leqslant 2^{32}$.

a) Evaluate Marsaglia's upper bound on the number of planes which will contain all consecutive $k = 10$ tuples from the MCG.

b) Repeat the previous part, but assume now a much larger bound $P \leqslant 2^{64}$.

c) Repeat the previous two parts for $k = 20$ and again for $k = 100$.

3.4. Suppose that an MCG becomes available with period $2^{19937} - 1$. What is Marsaglia's upper bound on the number of planes in $[0, 1]^{10}$ that will contain all 10-tuples from such a generator?

3.5. Consider the inversive generator $x_i = (a_0 + a_1 x_{i-1}^{-1}) \bmod p$ for $p = 59$, $a_0 = 1$ and $a_1 = 17$. Here x^{-1} satisfies $xx^{-1} \bmod p = 1$ for $x \neq 0$ and 0^{-1} is taken to be 0.

a) What is the period of this generator?

b) Plot the consecutive pairs (x_i, x_{i+1}).

3.6. Here we investigate whether the digits of π appear to be random.

a) Find the first 10,000 digits of $\pi - 3 \doteq .14159$ after the decimal point, and report how many of these are 0's, 1's and ... 9's. These digits are available in many places on the Internet.

b) Report briefly how you got them into a form suitable for computation. You might be able to do it within a file editor, or you might prefer to write your own short C or perl or other program to get the data in a form suitable for computing. Either list your program or describe your sequence of edits. Also: indicate which URL you got the π digits from. One time, it appeared that not all π listings on the web agree!

c) A χ^2 test for uniformity has test statistic

$$X^2 = \sum_{j=0}^{9} (E_j - O_j)^2 / E_j$$

where E_j is the expected number of occurences of digit j and O_j is the observed number. Report the value of X^2 for this data. Report also the p-value $\mathbb{P}(\chi^2_{(9)} \geqslant X^2)$ where $\chi^2_{(9)}$ denotes a chi-squared random variable with 9 degrees of freedom and X^2 is the value you computed. 9 degrees of freedom are appropriate here because there are 10 cells, and one df is lost because their probabilities sum to 1. No more are lost, because no parameters need to be estimated to compute the E_j.

d) Now take the first 10,000 (non-overlapping) pairs of digits past the decimal. There are 100 possible digit pairs. Count how many times each one appears, and print the results in a 10 by 10 matrix. Compute X^2 and the appropriate p-value for X^2. (Use 99 degrees of freedom.)

e) Split the first 1,000,000 digits after the decimal point into 1000 consecutive blocks of 1000 digits. For each block of 1000 digits compute the p-value using a $\chi^2_{(9)}$ distribution to test the uniformity of digit counts 0 through 9. What is the smallest p-value obtained? What is the largest? Produce a histogram of the 1000 p-values. If the digits are random, it should look like the uniform distribution on $[0, 1]$.

3.7. Here we make a simple test of the Mersenne Twister, using a trivial starting seed. The test requires the original MT19937 code, widely available on the internet, which comes with a function called `init_by_array`.

a) Seed the Mersenne Twister using `init_by_array` with $N = 1$ and the unsigned long vector of length N, called `init_key` in the source code, having a single entry `init_key[0]` equal to 0. Make a histogram of the first 10,000 $\mathbf{U}(0,1)$ sample values from this stream (using the function genrand_real3). Apply the χ^2 test of uniformity based on the number of these sample values falling into each of 100 bins $[(b-1)/100, b/100)$ for $b = 1,\dots,100$. Report the p-value for this test.

b) Continue the previous part, until you have obtained 10,000 p-values, each based on consecutive blocks of 10,000 sample values from the stream. Make a histogram of these p-values and report the p-value of this second level test.

This exercise can be done with the C language version of MT19937. If you use a translation into a different language, then indicate which one you have used.

3.8 (Use of streams)**.** Let U_i be a sequence of IID $\mathbf{U}(0,1)$ random variables for integers i, including $i \leqslant 0$. Let $T = \min\{i \geqslant 1 \mid \sum_{j=1}^{i} \sqrt{U_j} \geqslant 40\}$ be the first future time that a barrier is crossed. Similarly, let $S = \min\{i \geqslant 0 \mid \sum_{j=0}^{i} U_{-j}^2 \geqslant 20\}$ represent an event defined by the past and present at time 0. These events are separated by time $T + S$. (They are determined via $T + S + 1$ of the U_i.)

a) Estimate $\mu_{40,20} = \mathbb{E}(T+S)$ by Monte Carlo and give a confidence interval. Use $n = 1000$ simulations.

b) Replace the threshold 40 in the definition of T by 30. Estimate $\mu_{30,20} = \mathbb{E}(T+S)$ by Monte Carlo and give a confidence interval, using $n = 1000$ simulations. Explain how you ensure that the same past events are used for both $\mu_{40,20}$ and $\mu_{30,20}$. Explain how you ensure that the same future points are used in both estimates.

3.9. Suppose that U, V, and W are independent $\mathbf{U}(0,1)$ random variables. Show that $U \oplus V$ is independent of $U \oplus W$. [The $\oplus$ operation is defined in §3.7.]

3.10. Prove Proposition 3.1 on low order moments of functions of pairwise independent random vectors.

3.11 (Research)**.** Given n points $\boldsymbol{U}_i \sim \mathbf{U}(0,1)^d$ we can form $N = 2^n - 1$ pairwise independent random vectors $\boldsymbol{X}_1, \dots, \boldsymbol{X}_N$ by summing all non-empty sets of $\boldsymbol{U}_i$ modulo 1. For instance with $n = 3$ we get seven pairwise independent vectors $\boldsymbol{U}_1$, $\boldsymbol{U}_2$, $\boldsymbol{U}_3$, $\boldsymbol{U}_1 \oplus \boldsymbol{U}_2$, $\boldsymbol{U}_1 \oplus \boldsymbol{U}_3$, $\boldsymbol{U}_2 \oplus \boldsymbol{U}_3$, and $\boldsymbol{U}_1 \oplus \boldsymbol{U}_2 \oplus \boldsymbol{U}_3$. This scheme allows us to recycle about $\log_2(N)$ physical random vectors instead of about $\sqrt{2N}$ as required by combining pairs. For a function f we then use the average $\widetilde{Y}$ of N function values $f(\boldsymbol{U}_1), \dots, f(\boldsymbol{U}_1 \oplus \cdots \oplus \boldsymbol{U}_n)$ to estimate $\int_{(0,1)^d} f(\boldsymbol{u})\,\mathrm{d}\boldsymbol{u}$.

a) For $n = 8$, $d = 1$ and $f(X) = \exp(\Phi^{-1}(X))$ compute the distribution of $\widetilde{Y}$, the average of all $N = 255$ pairwise independent combinations. Do this by computing it 10,000 times using independent (pseudo)random $U_1, \dots, U_8 \sim \mathbf{U}(0,1)$ inputs each time. Show the histogram of the 10,000 resulting values $\widetilde{Y}_1, \dots, \widetilde{Y}_{10,000}$. Does it appear roughly symmetric?

b) Revisit the previous part, but now using $n = 23$ and averaging only the $N = \binom{23}{2} = 253$ pairwise sums of $U_1, \dots, U_n$ as described in §3.7. Show the histogram of the resulting values $\bar{Y}_1, \dots, \bar{Y}_{10,000}$ from equation (3.9). Does it appear roughly symmetric?

c) Compare the two methods graphically, by sorting their values so that $\widetilde{Y}_{(1)} \leqslant \widetilde{Y}_{(2)} \leqslant \cdots \leqslant \widetilde{Y}_{(10,000)}$ and $\bar{Y}_{(1)} \leqslant \bar{Y}_{(2)} \leqslant \cdots \leqslant \bar{Y}_{(10,000)}$ and plotting $\widetilde{Y}_{(r)}$ versus $\bar{Y}_{(r)}$. If the methods are roughly equivalent then these points should lie along a line through the point (μ, μ) (for $\mu = \exp(1/2)$) and slope $\sqrt{253/255}$. Do the methods appear to be roughly equivalent? What value do you get for $\sum_{r=1}^{10,000}(\widetilde{Y}_r - \mu)^4 / \sum_{r=1}^{10,000}(\bar{Y}_r - \mu)^4$?

The function Φ^{-1} is the inverse of the $\mathcal{N}(0,1)$ cumulative distribution function. This exercise requires a computing environment with an implementation of Φ^{-1}.

Bibliography

Daemen, J. and Rijmen, V. (2002). *The design of Rijndael: AES-The advanced encryption standard*. Springer, Berlin.

Gentle, J. E. (2003). *Random number generation and Monte Carlo methods*. Springer, New York, 2nd edition.

Haramoto, H., Matsumoto, M., Nishimura, T., Panneton, F., and L'Ecuyer, P. (2008). Efficient jump ahead for F_2-linear random number generators. *INFORMS Journal on Computing*, 20(3):385–390.

Hellekalek, P. (1998). Don't trust parallel Monte Carlo! In *Proceedings of the twelfth workshop on parallel and distributed simulation*, pages 82–89.

Hellekalek, P. and L'Ecuyer, P. (1998). Random number generators: selection criteria and testing. In Hellekalek, P. and Larcher, G., editors, *Random and quasi-random point sets*, pages 223–265. Springer, New York.

Knuth, D. E. (1998). *The Art of Computer Programming*, volume 2: Seminumerical algorithms. Addison-Wesley, Reading MA, 3rd edition.

L'Ecuyer, P. (1994). Uniform random number generation. *Annals of operations research*, 53(1):77–120.

L'Ecuyer, P. (1998). Random number generation. In Banks, J., editor, *Handbook on Simulation*. Wiley, New York.

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

L'Ecuyer, P. and Granger-Piché, J. (2003). Combined generators with components from different families. *Mathematics and computers in simulation*, 62:395–404.

L'Ecuyer, P. and Simard, R. (2007). TestU01: a C library for empirical testing of random number generators. *ACM transactions on mathematical software*, 33(4):article 22.

L'Ecuyer, P., Simard, R., Chen, E. J., and Kelton, W. D. (2002). An object-oriented random number package with many long streams and substreams. *Operations research*, 50(6):131–137.

Lehmer, D. H. (1951). Mathematical models in large-scale computing units. In *Proceedings of the second symposium on large scale digital computing machinery*, Cambridge MA. Harvard University Press.

Leopardi, P. (2009). Testing the tests: using random number generators to improve empirical tests. In L'Ecuyer, P. and Owen, A. B., editors, *Monte Carlo and Quasi-Monte Carlo Methods 2008*, pages 501–512, Berlin. Springer-Verlag.

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

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

Marsaglia, G. (1968). Random numbers fall mainly in the planes. *Proceedings of the National Academy of Science*, 61:25–28.

Marsaglia, G. and Tsang, W. W. (2002). Some difficult-to-pass tests of randomness. *Journal of Statistical Software*, 7(3):1–9.

Mascagni, M. and Srinivasan, A. (2004). Parametrizing parallel multiplicative lagged-Fibonacci generators. *ACM Transactions on Mathematical Software*, 30(7):899–916.

Matsumoto, M. and Kurita, Y. (1992). Twisted GFSR generators. *ACM transactions on modeling and computer simulation*, 2(3):179–194.

Matsumoto, M. and Nishimura, T. (1998). Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM transactions on modeling and computer simulation*, 8(1):3–30.

Matsumoto, M., Wada, I., Kuramoto, A., and Ashihara, H. (2007). Common defects in initialization of pseudorandom number generators. *ACM transactions on modeling and computer simulation*, 17(4).

Owen, A. B. (2009). Recycling physical random numbers. *Electronic Journal of Statistics*, 3:1531–1541.

Tausworthe, R. C. (1965). Random numbers generated by linear recurrence modulo two. *Mathematics of Computation*, 19(90):27–49.

Tezuka, S. (1995). *Uniform random numbers: theory and practice.* Kluwer Academic Publishers, Boston.