2

Uniform random numbers

All simulations work on a 'raw material' of *random numbers*. A sequence $R_1, R_2, \ldots$ is said to be a random number sequence if $R_i \sim U(0, 1)$ for all i and R_i is independent of R_j for all $i \neq j$. Some authors use the phrase 'random numbers' to include variates sampled from any specified probability distribution. However, its use here will be reserved solely for $U(0, 1)$ variates.

How can such a sequence be generated? One approach is to use a physical randomizing device such as a machine that picks lottery numbers in the UK, a roulette wheel, or an electronic circuit that delivers 'random noise'. There are two disadvantages to this. Firstly, such devices are slow and do not interface naturally with a computer. Secondly, and paradoxically, there is often a need to *reproduce* the random number stream (making it nonrandom!). This need arises, for example, when we wish to control the input to a simulation for the purpose of verifying the correctness of the programming code. It is also required when we wish to compare the effect of two or more policies on a simulation model. By using the same random number stream(s) for each of the experiments we hope to reduce the variance of estimators of the *difference* in response between any two policies.

One way to make a random number stream reproducible is to copy it to a peripheral and to read the numbers as required. A peripheral could be the hard disc of a computer, a CD ROM, or simply a book. In fact, the RAND corporation published 'A million random digits with 100 000 random normal deviates' (Rand Corporation, 1955), which, perhaps, was not that year's best seller. Accessing a peripheral several thousands or perhaps millions of times can slow down a simulation. Therefore, the preferred approach is to generate *pseudo-random numbers* at run-time, using a specified deterministic recurrence equation on integers. This allows fast generation, eliminates the storage problem, and gives a reproducible sequence. However, great care is needed in selecting an appropriate recurrence, to make the sequence *appear* random.

Simulation and Monte Carlo: With applications in finance and MCMC J. S. Dagpunar
© 2007 John Wiley & Sons, Ltd. ISBN: 978-0-470-85494-5

2.1 Linear congruential generators

These deliver a sequence of non-negative integers $\{X_i, i = 1, 2, \ldots\}$ where

$$X_i = (aX_{i-1} + c) \bmod m \quad (i = 1, 2, \ldots).$$

The recurrence uses four integer parameters set by the user. They are: $a(> 0)$ a *multiplier*, $X_0(\geq 0)$ a *seed*, $c(\geq 0)$ an *increment*, and $m(>0)$ a *modulus*. The first three parameter values lie in the interval $[0, m-1]$. The modulo m process returns the remainder after dividing $aX_{i-1} + c$ by m. Therefore, $X_i \in [0, m-1]$ for all i. The pseudo-random number is delivered as $R_i = X_i/m$ and so $R_i \in [0, 1)$. The idea is that if m is large enough, the discrete values $\frac{0}{m}, \frac{1}{m}, \frac{2}{m}, \ldots, \frac{m-1}{m}$ are so close together that R_i can be treated as a continuously distributed random variable. In practice we do not use $X_i = 0\,(R_i = 0)$ in order to avoid problems of division by zero and taking the logarithm of zero, etc. As an example consider the generator

$$X_i = (9X_{i-1} + 3) \bmod 2^4 \quad (i = 1, 2, \ldots). \tag{2.1}$$

Choose $X_0 \in [0, 15]$, say $X_0 = 3$. Then $X_1 = 30 \bmod 2^4 = 14$, $X_2 = 129 \bmod 2^4 = 1, \ldots$. The following sequence for $\{R_i\}$ is obtained:

$$\frac{3}{16}, \frac{14}{16}, \frac{1}{16}, \frac{12}{16}, \frac{15}{16}, \frac{10}{16}, \frac{13}{16}, \frac{8}{16}, \frac{11}{16}, \frac{6}{16}, \frac{9}{16}, \frac{4}{16}, \frac{7}{16}, \frac{2}{16}, \frac{5}{16}, \frac{0}{16}, \frac{3}{16}, \ldots. \tag{2.2}$$

The *period* of a generator is the smallest integer λ such that $X_\lambda = X_0$. Here the sequence repeats itself on the seventeenth number and so $\lambda = 16$. Clearly, we wish to make the period as large as possible to avoid the possibility of reusing random numbers. Since the period cannot exceed m, the modulus is often chosen to be close to the largest representable integer on the computer.

There are some number theory results (Hull and Dobell, 1962) that assist in the choice of a, c, and m. A *full period* $(=m)$ is obtained if and only if

c and *m are relatively prime* (*greatest common divisor of* c and m is 1); (2.3)

$a - 1$ is a multiple of q for every prime factor q of m; (2.4)

$a - 1$ is a multiple of 4 if m is. (2.5)

Linear congruential generators can be classified into *mixed* $(c > 0)$ and *multiplicative* $(c = 0)$ types.

2.1.1 Mixed linear congruential generators

In this case $c > 0$. A good choice is $m = 2^b$ where b is the maximum number of bits used to represent positive integers with a particular computer/language combination. For example, many computers have a 32 bit word for integers. One bit may be reserved for the sign, leaving $b = 31$. Such a computer can store integers in the interval $\left[-2^{31}, 2^{31} - 1\right]$.

By choosing $m = 2^b$ the generator will have a full period ($\lambda = m$) when c is chosen to be odd-valued, satisfying condition (2.3), and $a - 1$ to be a multiple of 4, satisfying conditions (2.4) and (2.5). This explains why the generator (2.1) is a full period one with $\lambda = 16$.

In Maple there is no danger that $aX_{i-1} + c$ exceeds N, the largest positive integer that can be stored on the computer. This is because Maple performs arithmetic in which the number of digits in N is essentially limited only by the memory available to Maple. The Maple procedure 'r1' below uses parameter values $m = 2^{31}$, $a = 906185749$, $c = 1$. Note the importance of declaring 'seed' as a global variable, so that the value of this variable can be transmitted to and from the procedure. The period of the generator is $m = 2147483648$, which is adequate for most simulations. This generator has been shown to have good statistical properties (Borosh and Niederreiter, 1983). More will be stated about statistical properties in Sections 2.2 and 2.4.

```
> r1 := proc() global seed;
  seed:= (906185749*seed + 1) mod 2^31;
  evalf(seed/2^31);
  end proc:
```

The code below invokes the procedure five times starting with seed = 3456:

```
> seed:= 3456;
  for j from 1 to 5 do;
  r1();
  end do;

seed:= 3456
.3477510815
.2143113120
.7410933147
.4770359378
.6231261701
```

The generator took about 12 microseconds per random number using a Pentium M 730 processor. The generator 'r2' below (L'Ecuyer, 1999) has a far more impressive period. The parameter values are $m = 2^{64}$, $a = 2,862,933,555,777,941,757$, $c = 1$. The period is 18, 446, 744, 073, 709, 551, 616 and the execution speed is not much slower at 16 microseconds per random number.

```
> r2:= proc() global seed;
  seed:= (2862933555777941757*seed + 1) mod 2^64;
  evalf(seed/2^64);
  end proc:
```

In most scientific languages (e.g. FORTRAN90, C++) the finite value of N is an issue. For example, N may be $2^{31} - 1$ when the word size is 32 bits. Explicit calculation of $aX_{i-1} + c$ may cause a fatal error through *overflow* when $aX_{i-1} + c \notin \left[-2^{31}, 2^{31} - 1\right]$. However, in some implementations, the integer following N is stored as $-(N+1)$

followed by $-N$, and so overflow never occurs. In that case, it is necessary only to take the last 31 bits. In such cases the modulo m process is particularly simple to implement. If this feature is not present, overflow can be avoided by working in *double precision*. For example, with a 32 bit word size, double precision uses 64 bits. However, the random number generator will be somewhat slower than with single precision.

Another way of dealing with potential overflow will now be described. It is based on a method due to Schrage (1979). Let

$$u = \left\lfloor \frac{m}{a} \right\rfloor, \quad w = m \bmod a.$$

Then

$$m = ua + w.$$

Now,

$$X_i = \left\lfloor \frac{X_i}{u} \right\rfloor u + X_i \bmod u$$

and so

$$\begin{aligned} aX_i + c &= \left\lfloor \frac{X_i}{u} \right\rfloor au + a\left(X_i \bmod u\right) + c \\ &= \left\lfloor \frac{X_i}{u} \right\rfloor (m - w) + a\left(X_i \bmod u\right) + c. \end{aligned}$$

Therefore,

$$\left(aX_i + c\right) \bmod m = \left\{ -\left\lfloor \frac{X_i}{u} \right\rfloor w + a\left(X_i \bmod u\right) + c \right\} \bmod m. \tag{2.6}$$

Now

$$0 \le \left\lfloor \frac{X_i}{u} \right\rfloor w \le \frac{X_i w}{u} \le X_i \le m - 1$$

providing that a is chosen such that

$$w \le u. \tag{2.7}$$

Similarly,

$$\begin{aligned} & c \le a\left(X_i \bmod u\right) + c \le a\left(u - 1\right) + c \\ & = m - w - a + c \\ & \le m - 1 \end{aligned}$$

providing a and c are chosen such that

$$m - a \ge w \ge c + 1 - a. \tag{2.8}$$

Therefore, subject to conditions (2.7) and (2.8)

$$-m+1+c \le -\left\lfloor \frac{X_i}{u} \right\rfloor w + a\,(X_i \bmod u) + c \le m-1.$$

To perform the mod m process in Equation (2.6) simply set

$$Z_{i+1} = -\left\lfloor \frac{X_i}{u} \right\rfloor w + [a\,(X_i \bmod u)] + c.$$

Then

$$X_{i+1} = \begin{cases} Z_{i+1} & (Z_i \ge 0)\,, \\ Z_{i+1} + m & (Z_i < 0)\,. \end{cases}$$

The Maple procedure below, named 'schrage', implements this for the full period generator with $m = 2^{32}$, $a = 69069$, and $c = 1$. It is left as an exercise (see Problem 2.3) to verify the correctness of the algorithm, and in particular that conditions (2.7) and (2.8) are satisfied. It is easily recoded in any scientific language. In practice, it would not be used in a Maple environment, since the algorithm is of most benefit when the maximum allowable size of a positive integer is 2^{32}. The original generator with these parameter values, but without the Schrage innovation, is a famous one, part of the 'SUPER-DUPER' random number suite (Marsaglia, 1972; Marsaglia *et al.*, 1972). Its statistical properties are quite good and have been investigated by Anderson (1990) and Marsaglia and Zaman (1993).

```
> schrage:=proc() local s,r; global seed;
  s:=seed mod 62183;r:= (seed-s)/62183;
  seed:=-49669*r+69069*s+1;
  if seed< 0 then seed:=seed+2^32 end if;
  evalf(seed/2^32);
  end proc;
```

Many random number generators are proprietary ones that have been coded in a lower level language where the individual bits can be manipulated. In this case there is a definite advantage in using a modulus of $m = 2^b$. The evaluation of $(aX_{i-1} + c) \bmod 2^b$ is particularly efficient since X_i is returned as the last b bits of $aX_{i-1} + c$. For example, in the generator (2.1)

$$X_7 = (9 \times 13 + 3)(\text{mod } 16).$$

In binary arithmetic, $X_7 = [(1001.) \times (1101.) + (11.)]$ mod 10000. . Now $(1001.) \times (1101.) + (11.) =$

$$\begin{array}{ll} 0110\,1000. & \\ 0000\,1101. & + \\ \underline{0000\,0011.} & \end{array} \tag{2.9}$$

$$\underline{0111\,1000.} \tag{2.10}$$

Note that the first row of (2.9) gives (1000.) $\times$ (1101.) by shifting the binary (as opposed to the decimal) point of (1101.) 3 bits to the right. The second row gives (0001.) $\times$ (1101.) and the third row is (11.). The sum of the three rows is shown in (2.10). Then X_7 is the *final* 4 bits in this row, that is 1000., or $X_7 = 8$ in the decimal system. In fact, it is unnecessary to perform any calculations beyond the fourth bit. With this convention, and omitting bits other than the last four, $X_8 = \{(1001.) \times (1000.) + (11.)\}$ mod (10000.) =

$$\begin{array}{ll} 0\,0\,0\,0. & \\ 1\,0\,0\,0. & + \\ \underline{0\,0\,1\,1.} & \\ \underline{1\,0\,1\,1.} & \end{array}$$

or $X_8 = 1011.$ (binary), or $X_8 = 11$ (decimal). To obtain R_8 we divide by 16. In binary, this is done by moving the binary point four bits to the left, giving (0.1011) or $2^{-1} + 2^{-3} + 2^{-4} = 11/16$. By manipulating the bits in this manner the issue of overflow does not arise and the generator will be faster than one programmed in a high-level language. This is of benefit if millions of numbers are to be generated.

2.1.2 Multiplicative linear congruential generators

In this case $c = 0$. This gives

$$X_i = (aX_{i-1}) \bmod m.$$

We can never allow $X_i = 0$, otherwise the subsequent sequence will be …, 0,0,…. Therefore the period cannot exceed $m - 1$. Similarly, the case $a = 1$ can be excluded. It turns out that a maximum period of $m - 1$ is achievable if and only if

$$m \text{ is prime and} \tag{2.11}$$

$$a \text{ is a primitive root of } m \tag{2.12}$$

A multiplicative generator satisfying these two conditions is called a *maximum period prime modulus generator*. Requirement (2.12) means that

$$m \nmid a \text{ and } m \nmid a^{(m-1)/q} - 1 \text{ for every prime factor } q \text{ of } m - 1. \tag{2.13}$$

Since the multiplier a is always chosen such that $a < m$, the first part of this condition can be ignored. The procedure ‘r3’ shown below is a good (see Section 2.2) maximum period prime modulus generator with multiplier $a = 630360016$. It takes approximately 11 microseconds to deliver one random number using a Pentium M 730 processor:

```
> r3:= proc() global seed;
  seed:= (seed*630360016)mod(2^31 − 1);
  evalf(seed/(2^31 − 1));
  end proc;
```

At this point we will describe the in-built Maple random number generator, 'rand()' (Karian and Goyal, 1994). It is a maximum period prime modulus generator with $m = 10^{12} - 11$, $a = 427419669081$ (Entacher, 2000). To return a number in the interval [0,1), we divide by $10^{12} - 11$, although it is excusable to simply divide by 10^{12}. It is slightly slower than 'r1' (12 microseconds) and 'r2' (16 microseconds), taking approximately 17 microseconds per random number. The seed is set using the command 'randomize(integer)' before invoking 'rand()'. Maple also provides another $U(0, 1)$ generator. This is 'stats[random,uniform] (1)'. This is based upon 'rand' and so it is surprising that its speed is approximately 1/17th of the speed of 'rand()/10^12'. It is not advised to use this.

For any prime modulus generator, $m \neq 2^b$, so we cannot simply deliver the last b bits of aX_{i-1} expressed in binary. Suppose $m = 2^b - \gamma$ where γ is the smallest integer that makes m prime for given b. The following method (Fishman, 1978, pp. 357–358) *emulates* the bit shifting process, previously described for the case $m = 2^b$. The generator is

$$X_{i+1} = (aX_i) \bmod (2^b - \gamma).$$

Let

$$Y_{i+1} = (aX_i) \bmod 2^b, \quad K_{i+1} = \lfloor aX_i/2^b \rfloor . \tag{2.14}$$

Then

$$aX_i = K_{i+1}2^b + Y_{i+1}.$$

Therefore,

$$\begin{aligned} X_{i+1} &= \left(K_{i+1}2^b + Y_{i+1}\right) \bmod (2^b - \gamma) \\ &= \left\{K_{i+1}(2^b - \gamma) + Y_{i+1} + \gamma K_{i+1}\right\} \bmod (2^b - \gamma) \\ &= \{Y_{i+1} + \gamma K_{i+1}\} \bmod (2^b - \gamma) \end{aligned}$$

From (2.14), $0 \le Y_{i+1} \le 2^b - 1$ and $0 \le K_{i+1} \le \lfloor a(2^b - \gamma - 1)/2^b \rfloor \le a - 1$. Therefore, $0 \le Y_{i+1} + \gamma K_{i+1} \le 2^b - 1 + a\gamma - \gamma$. We would like $Y_{i+1} + \gamma K_{i+1}$ to be less than $2^b - \gamma$ so that it may be assigned to X_{i+1} without performing the troublesome $\bmod(2^b - \gamma)$. Failing that, it would be convenient if it was less than $2(2^b - \gamma)$, so that $X_{i+1} = \{Y_{i+1} + \gamma K_{i+1}\} - (2^b - \gamma)$, again avoiding the $\bmod(2^b - \gamma)$ process. This will be the case if $2^b - 1 + a\gamma - \gamma \le 2(2^b - \gamma) - 1$, that is if

$$a \le \frac{2^b}{\gamma} - 1. \tag{2.15}$$

In that case, set $Z_{i+1} = Y_{i+1} + \gamma K_{i+1}$. Then

$$X_{i+1} = \begin{cases} Z_{i+1} & \left(Z_{i+1} < 2^b - \gamma\right), \\ Z_{i+1} - (2^b - \gamma) & \left(Z_{i+1} \ge 2^b + \gamma\right). \end{cases}$$

The case $\gamma = 1$ is of practical importance. The condition (2.15) reduces to $a \le 2^b - 1$. Since $m = 2^b - \gamma = 2^b - 1$, the largest possible value that could be chosen for a is $2^b - 2$. Therefore, when $\gamma = 1$ the condition (2.15) is satisfied for *all* multipliers a. Prime numbers of the form $2^k - 1$ are called *Mersenne primes*, the low-order ones being $k = 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, \ldots$

How do we find primitive roots of a prime, m? If a is a primitive root it turns out that the others are

$$\left\{a^j \bmod m : j < m-1, j \text{ and } m-1 \text{ are relatively prime}\right\}. \tag{2.16}$$

As an example we will construct all maximum period prime modulus generators of the form

$$X_{i+1} = (aX_i) \bmod 7.$$

We require all primitive roots of 7 and refer to the second part of condition (2.13). The prime factors of $m - 1 = 6$ are 2 and 3. If $a = 2$ then $7 = m \mid a^{6/2} - 1 = 2^{6/2} - 1$. Therefore, 2 is not a primitive root of 7. If $a = 3, 7 = m \nmid a^{6/2} - 1 = 3^{6/2} - 1$ and $7 = m \nmid a^{6/3} - 1 = 3^{6/3} - 1$. Thus, $a = 3$ is a primitive root of 7. The only j $(< m - 1)$ which is relatively prime to $m - 1 = 6$ is $j = 5$. Therefore, by (2.16), the remaining primitive root is $a = 3^5 \bmod m = 9 \times 9 \times 3 \bmod 7 = 2 \times 2 \times 3 \bmod 7 = 5$. The corresponding sequences are shown below. Each one is a reversed version of the other:

$$a = 3 : \ldots, \frac{1}{7}, \frac{3}{7}, \frac{2}{7}, \frac{6}{7}, \frac{4}{7}, \frac{5}{7}, \frac{1}{7}, \ldots$$
$$a = 5 : \ldots, \frac{1}{7}, \frac{5}{7}, \frac{4}{7}, \frac{6}{7}, \frac{2}{7}, \frac{3}{7}, \frac{1}{7}, \ldots$$

For larger moduli, finding primitive roots by hand is not very easy. However, it is easier with Maple. Suppose we wish to construct a maximum period prime modulus generator using the Mersenne prime $m = 2^{31} - 1$ and would like the multiplier $a \approx m/2 = 1073741824.5$. All primitive roots can be found within, say, 10 of this number using the code below:

```
> with(numtheory):
  a:=1073741814;
  do;
  a:=primroot(a, 2^31-1);
  if a > 1073741834 then break end if;
  end do;
```

```
a := 1073741814
a := 1073741815
a := 1073741816
a := 1073741817
a := 1073741827
a := 1073741829
a := 1073741839
```

We have concentrated mainly on the maximum period prime modulus generator, because of its almost ideal period. Another choice will briefly be mentioned where the modulus is $m = 2^b$ and b is the usable word length of the computer. In this case the maximum period achievable is $\lambda = m/4$. This occurs when $a = 3 \bmod 8$ or $5 \bmod 8$, and X_0 is odd. In each case the sequence consists of $m/4$ odd numbers, which does not communicate with the other sequence comprising the remaining $m/4$ odd numbers. For example, $X_i = (3X_{i-1}) \bmod 2^4$ gives either

$$\ldots, \frac{1}{15}, \frac{3}{15}, \frac{9}{15}, \frac{11}{15}, \frac{1}{15}, \ldots$$

or

$$\ldots, \frac{5}{15}, \frac{15}{15}, \frac{13}{15}, \frac{7}{15}, \frac{5}{15}, \ldots$$

depending upon the choice of seed.

All Maple procedures in this book use 'rand' described previously. Appendix 2 contains the other generators described in this section.

2.2 Theoretical tests for random numbers

Most linear congruential generators are one of the following three types, where λ is the period:

Type A: full period multiplicative, $m = 2^b, a = 1 \bmod 4$, c odd-valued, $\lambda = m$;
Type B: maximum period multiplicative prime modulus, m a prime number, $a =$ a primitive root of m, $c = 0$, $\lambda = m - 1$;
Type C: maximum period multiplicative, $m = 2^b$, $a = 5 \bmod 8$, $\lambda = m/4$.

The output from type C generators is *identical* (apart from the subtraction of a specified constant) to that of a corresponding type A generator, as the following theorem shows.

Theorem 2.1 *Let* $m = 2^b, a = 5 \bmod 8$, X_0 *be odd-valued,* $X_{i+1} = (aX_i) \bmod m$, $R_i = X_i/m$. *Then* $R_i = R_i^* + (X_0 \bmod 4)/m$ *where* $R_i^* = X_i^*/(m/4)$ *and* $X_{i+1}^* = (aX_i^* + [X_0 \bmod 4]\{(a-1)/4\}) \bmod (m/4)$.

Proof. First we show that $X_i - X_0 \bmod 4$ is a multiple of 4. Assume that this is true for $i = k$. Then $X_{k+1} - X_0 \bmod 4 = [a(X_k - X_0 \bmod 4) + (a-1)\{X_0 \bmod 4\}] \bmod m$. Now, $4 \mid a-1$ so $4 \mid X_{k+1} - X_0 \bmod 4$. For the base case $i = 0$, $X_i - X_0 \bmod 4 = 0$, and so by the principle of induction $4 \mid X_i - X_0 \bmod 4\ \forall i \geq 0$. Now put $X_i^* = (X_i - X_0 \bmod 4)/4$. Then $X_i = 4X_i^* + X_0 \bmod 4$ and $X_{i+1} = 4X_{i+1}^* + X_0 \bmod 4$. Dividing the former equation through by m gives $R_i = R_i^* + (X_0 \bmod 4)/m$ where $X_{i+1} - aX_i = 4\left(X_{i+1}^* - aX_i^*\right) - (X_0 \bmod 4)(a-1) = 0 \bmod m$. It follows that $X_{i+1}^* - aX_i^* - [X_0 \bmod 4][(a-1)/4] = 0 \bmod (m/4)$ since $4 \mid m$. This completes the proof.

This result allows the investigation to be confined to the theoretical properties of type A and B generators only. Theoretical tests use the values a, c, and m to assess the quality of the output of the generator over the *entire* period. It is easy to show (see Problem 5) for both type A and B generators that for all but small values of the period λ, the mean and variance of $\{R_i, i=0, \ldots, \lambda-1\}$ are close to $\frac{1}{2}$ and $\frac{1}{12}$, as must be the case for a true $U(0, 1)$ random variable.

Investigation of the *lattice* (Ripley, 1983a) of a generator affords a deeper insight into the quality. Let $\{R_i, i=0, \ldots, \lambda-1\}$ be the entire sequence of the generator. In theory it would be possible to plot the λ *overlapping pairs* $(R_0, R_1), \ldots, (R_{\lambda-1}, R_0)$. A necessary condition that the sequence consists of *independent* $U(0, 1)$ random variables is that R_1 is independent of R_0, R_2 is independent of R_1, and so on. Therefore, the pairs should be uniformly distributed over $[0, 1)^2$. Figures 2.1 and 2.2 show plots of 256 such points for the full period generators $X_{i+1}=(5X_i+3) \bmod 256$ and $X_{i+1}=(13X_i+3) \bmod 256$ respectively. Firstly, a disturbing feature of both plots is observed; all points can be covered by a set of parallel lines. This detracts from the uniformity over $[0, 1)^2$. However, it is unavoidable (for all linear congruential generators) given the linearity (mod m) of these recurrences. Secondly, Figure 2.2 is preferred in respect of uniformity over $[0, 1)^2$. The minimum number of lines required to cover all points is 13 in Figure 2.2 but only 5 in Figure 2.1, leading to a markedly nonuniform density of points in the latter case. The separation between adjacent lines is wider in Figure 2.1 than it is in Figure 2.2. Finally, each lattice can be constructed from a reduced basis consisting of vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ which define the smallest lattice cell. In Figure 2.1 this is long and thin, while in the more favourable case of Figure 2.2 the sides have similar lengths. Let $l_1=|\mathbf{e}_1|$ and $l_2=|\mathbf{e}_2|$ be the lengths of the smaller and longer sides respectively. The larger $r_2=l_2/l_1$ is, the poorer the uniformity of pairs and the poorer the generator.

This idea can be extended to find the degree of uniformity of the set of overlapping k-tuples $\{(R_i, \ldots, R_{i+k-1 \bmod m}), i=0, \ldots, \lambda-1\}$ through the hypercube $[0, 1)^k$. Let $l_1, \ldots, l_k$ be the lengths of the vectors in the reduced basis with $l_1 \leq \cdots \leq l_k$. Alternatively, these are the side lengths of the smallest lattice cell. Then, generators for which $r_k=l_k/l_1$ is large, at least for small values of k are to be regarded with suspicion. Given values for a, c, and m, it is possible to devise an algorithm that will calculate either r_k or an upper bound for r_k (Ripley, 1983a). It transpires that changing the value of c in a type A generator only translates the lattice as a whole; the relative positions of the lattice points remain unchanged. As a result the choice of c is immaterial to the quality of a type A generator and the crucial decision is the choice of a.

Table 2.1 gives some random number generators that are thought to perform well. The first four are from Ripley (1983b) and give good values for the lattice in low dimensions. The last five are recommended by (Fishman and Moore 1986) from a search over all multipliers for prime modulus generators with modulus $2^{31}-1$. There are references to many more random number generators with given parameter values together with the results of theoretical tests in Entacher (2000).

2.2.1 Problems of increasing dimension

Consider a maximum period multiplicative generator with modulus $m \approx 2^b$ and period $m-1$. The random number sequence is a permutation of $1/m, \ldots, (m-1)/m$. The distance between neighbouring values is constant and equals $1/m$. For sufficiently large m this is small enough to ignore the 'graininess' of the sequence. Consequently, we are happy to use this discrete uniform as an approximation to a continuous

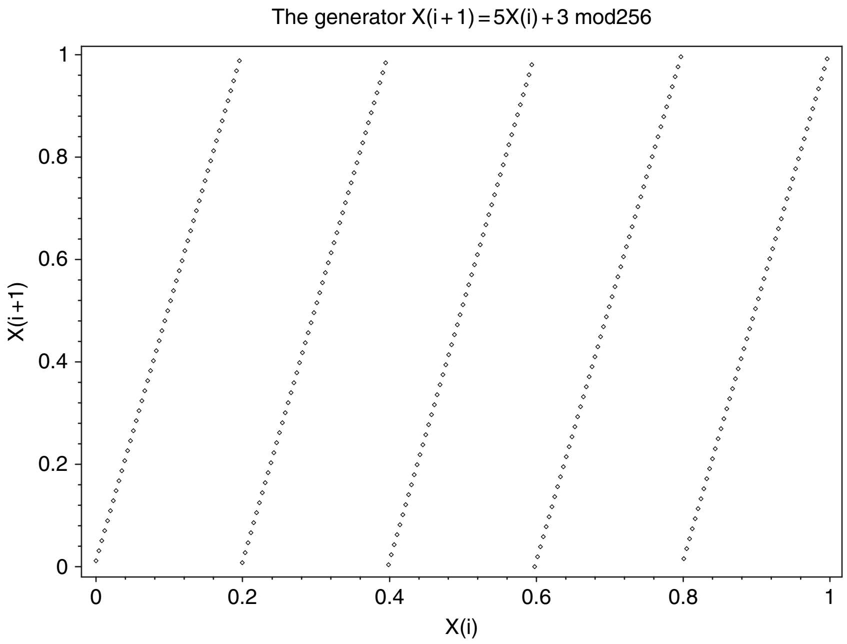

Figure 2.1 Plot of $\{(X_i, X_{i+1}), i = 0, \ldots, 255\}$ for $X_{i+1} = (5X_i + 3) \bmod 256$

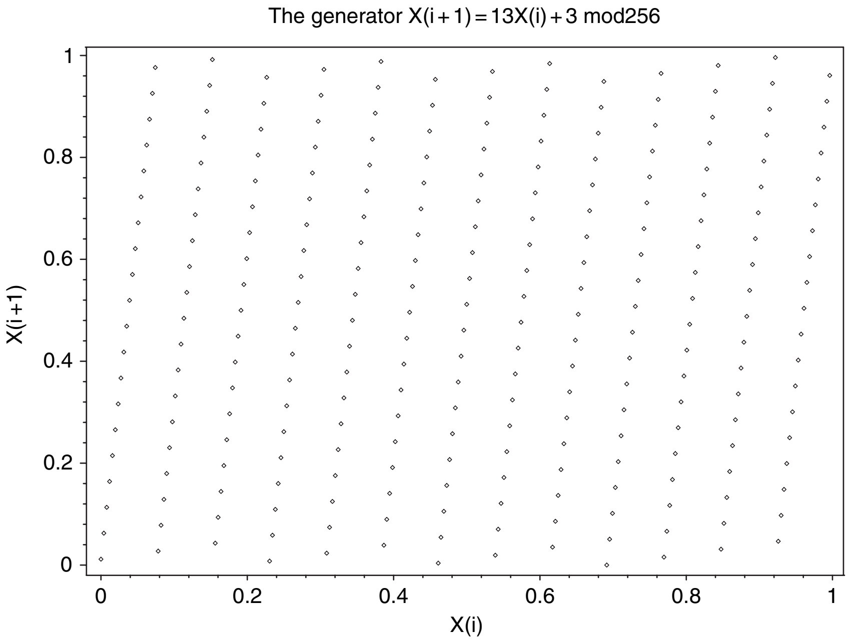

Figure 2.2 Plot of $\{(X_i, X_{i+1}), i = 0, \ldots, 255\}$ for $X_{i+1} = (13X_i + 3) \bmod 256$

Table 2.1 Some recommended linear congruential generators. (Data are from Ripley, 1983b and Fishman and Moore, 1986)

m	a	c	r_2	r_3	r_4
2^{59}	13^{13}	0	1.23	1.57	1.93
2^{32}	69069	Odd	1.06	1.29	1.30
$2^{31}-1$	630360016	0	1.29	2.92	1.64
2^{16}	293	Odd	1.20	1.07	1.45
$2^{31}-1$	950,706,376	0			
$2^{31}-1$	742,938,285	0			
$2^{31}-1$	1,226,874,159	0			
$2^{31}-1$	62,089,911	0			
$2^{31}-1$	1,343,714,438	0			

$U(0,1)$ whenever b-bit accuracy of a continuous $U[0,1)$ number suffices. In order to generate a point uniformly distributed over $[0,1)^2$ we would usually take two *consecutive* random numbers. There are $m-1$ such 2-tuples or points. However, the average distance of a 2-tuple to its nearest neighbour (assuming r_2 is close to 1) is approximately $1/\sqrt{m}$. In k dimensions the corresponding distance is approximately $1/\sqrt[k]{m} = 2^{-b/k}$, again assuming an ideal generator in which r_k does not differ too much from 1. For example, with $b = 32$, an integral in eight dimensions will be approximated by the expectation of a function of a random vector having a discrete (rather than the desired continuous) distribution in which the average distance to a nearest neighbour is of the order of $2^{-4} = \frac{1}{16}$. In that case the graininess of the discrete approximation to the continuous uniform distribution might become an issue. One way to mitigate this effect is to *shuffle* the output in order that the number of possible k-tuples is much geater than the $m-1$ that are available in an unshuffled sequence. Such a method is described in Section 2.3. Another way to make the period larger is to use Tauseworthe generators (Tauseworthe, 1965; Toothill *et al.*, 1971; Lewis and Payne, 1973; Toothill *et al.*, 1973). Another way is to combine generators. An example is given in Section 2.5. All these approaches can produce sequences with very large periods. A drawback is that their theoretical properties are not so well understood as the output from a standard unshuffled linear congruential generator. This perhaps explains why the latter are in such common usage.

2.3 Shuffled generator

One way to break up the lattice structure is to permute or shuffle the output from a linear congruential generator. A *shuffled generator* works in the following way. Consider a generator producing a sequence $\{U_1, U_2, \ldots\}$ of $U(0,1)$ numbers. Fill an array $T(0), T(1), \ldots, T(k)$ with the first $k+1$ numbers $U_1, \ldots, U_{k+1}$. Use $T(k)$ to determine a number N that is $U[0, k-1]$. Output $T(k)$ as the next number in the shuffled sequence. Replace $T(k)$ by $T(N)$ and then replace $T(N)$ by the next random number U_{k+2} in the un-shuffled sequence. Repeat as necessary. An algorithm for this is:

```
$N := \lfloor kT(k) \rfloor$
*Output* $T(k)$ (becomes the next number in the shuffled sequence)
$T(k) := T(N)$
*Input* $U$ (the next number from the unshuffled sequence)
$T(N) := U$
```

Note that $\lfloor x \rfloor$ denotes the floor of x. Since x is non-negative, it is the integer part of x. An advantage of a shuffled generator is that the period is increased.

2.4 Empirical tests

Empirical tests take a segment of the output and subject it to statistical tests to determine whether there are specific departures from randomness.

2.4.1 Frequency test

Here we test the hypothesis that $R_i, i = 1, 2, \ldots,$ are uniformly distributed in (0,1). The test assumes that $R_i, i = 1, 2, \ldots,$ are *independently* distributed. We take n consecutive numbers, $R_1, \ldots, R_n$, from the generator. Now divide the interval $(0, 1)$ into k subintervals $(0, h), [h, 2h), \ldots, [\{k-1\}h, kh)$ where $kh = 1$. Let f_i denote the observed frequency of observations in the ith subinterval. We test the null hypothesis that the sample is from the $U(0, 1)$ distribution against the alternative that it is not. Let $e_i = n/k$, which is the expected frequency assuming the null hypothesis is true. Under the null hypothesis the test statistic

$$X^2 = \sum_{i=1}^{k} \frac{(f_i - e_i)^2}{e_i} = \sum_{i=1}^{k} \frac{f_i^2}{e_i} - n$$

follows a chi-squared distribution with $k-1$ degrees of freedom. Large values of X^2 suggest nonuniformity. Therefore, the null hypothesis is rejected at the $100\alpha\,\%$ significance level if $X^2 > \chi^2_{k-1,\alpha}$ where $\alpha = P\left(\chi^2_{k-1} > \chi^2_{k-1,\alpha}\right)$.

As an example of this, 1000 random numbers were sampled using the Maple random number generator, 'rand()'. Table 2.2 gives the observed and expected frequencies based upon $k = 10$ subintervals of width $h = 0.1$. This gives

$$X^2 = \frac{100676}{100} - 1000 = 6.76.$$

From tables of the percentage points of the chi-squared distribution it is found that $\chi^2_{9,0.05} = 16.92$, indicating that the result is not significant. Therefore, there is insufficient evidence to dispute the uniformity of the population, assuming that the observations are independent.

The null chi-squared distribution is an asymptotic result, so the test can be applied only when n is suitably large. A rule of thumb is that $e_i \gtrapprox 5$ for every interval. For a really large sample we can afford to make k large. In that case tables for chi-squared are

Table 2.2 Observed and expected frequencies

Interval	$[0, h)$	$[h, 2h)$	$[2h, 3h)$	$[3h, 4h)$	$[4h, 5h)$	$[5h, 6h)$	$[6h, 7h)$	$[7h, 8h)$	$[8h, 9h)$	$[9h, 1)$
f_i	99	94	95	108	108	88	111	92	111	94
e_i	100	100	100	100	100	100	100	100	100	100

not available, but the asymptotic normality of the distribution can be used: as $m \to \infty$, $\sqrt{2\chi^2_m} \to N\left(\sqrt{2m-1}, 1\right)$.

A disadvantage of the chi-squared test is that it is first necessary to test for the *independence* of the random variables, and, secondly, by dividing the domain into intervals, we are essentially testing against a discrete rather than a continuous uniform distribution. The test is adequate for a crude indication of uniformity. The reader is referred to the Kolmogorov–Smirnov test for a more powerful test.

2.4.2 Serial test

Consider a sequence of random numbers $R_1, R_2, \ldots$. Assuming uniformity and independence, the *nonoverlapping* pairs $(R_1, R_2), (R_3, R_4), \ldots$ should be uniformly and independently distributed over $(0, 1)^2$. If there is serial dependence between consecutive numbers, this will be manifested as a clustering of points and the uniformity will be lost. Therefore, to investigate the possibility of serial dependence the null hypothesis that the pairs $(R_{2i-1}, R_{2i}), i = 1, 2, \ldots,$ are uniformly distributed over $(0, 1)^2$ can be tested against the alternative hypothesis that they are not. The chi-squared test can be applied, this time with k^2 subsquares each of area $1/k^2$. Nonoverlapping pairs are prescribed, as the chi-squared test demands independence. Let f_i denote the observed frequency in the ith subsquare, where $i = 1, 2, \ldots, k^2$, with $\sum_{i=1}^{k^2} f_i = n$, that is $2n$ random numbers in the sample. Under the null hypothesis, $e_i = n/k^2$ and the null distribution is $\chi^2_{k^2-1}$.

The assumption of independence between the n points in the sample is problematic, just as the assumption of independence between random numbers was in the frequency test. In both cases it may help to sample random numbers (points) that are not consecutive, but are some (random) distance apart in the generation scheme. Clearly, this serial correlation test is an empirical version of lattice tests. It can be extended to three and higher dimensions, but then the sample size will have to increase exponentially with the dimension to ensure that the expected frequency is at least five in each cell.

2.4.3 Other empirical tests

There is no limit to the number and type of empirical tests that can be devised, and it will always be possible to construct a test that will yield a statistically significant result in respect of some aspect of dependence. This must be the case since, among other reasons, the stream is the result of a deterministic recurrence. The important point is that 'gross' forms of nonrandomness are not present. It may be asked what constitutes 'gross'? It might be defined as those forms of dependence (or nonuniformity) that are detrimental to a particular Monte Carlo application. For example, in a k-dimensional definite integration, it is the uniformity of k-tuples $(R_i, R_{i+1}, \ldots, R_{i+k-1})$, $i = 0, k, 2k, \ldots,$ that is important. In practice, the user of random numbers rarely has the time or inclination to check these aspects. Therefore, one must rely on random number generators that have been thoroughly investigated in the literature and have passed a battery of theoretical and empirical tests. Examples of empirical (statistical) tests are the gap test, poker test, coupon collector's test, collision test, runs test, and test of linear dependence. These and a fuller description of generating and testing random numbers appear in Knuth (1998) and Dagpunar (1988a). The Internet server 'Plab' is devoted to research on random number generation at the Mathematics Department, Salzburg University. It is a useful source of generation methods and tests and is located at http://random.mat.sbg.ac.at/.

2.5 Combinations of generators

By combining the output from several independent generators it is hoped to (a) increase the period and (b) improve the randomness of the output. Aspect (a) is of relevance when working in high dimensions. A generator developed by Wichman and Hill (1982, 1984) combines the output of three congruential generators:

$$X_{i+1} = (171X_i) \bmod 30269,$$
$$Y_{i+1} = (172Y_i) \bmod 30307,$$
$$Z_{i+1} = (170Z_i) \bmod 30323.$$

Now define

$$R_{i+1} = \left(\frac{X_{i+1}}{30269} + \frac{Y_{i+1}}{30307} + \frac{Z_{i+1}}{30323} \right) \bmod 1 \tag{2.17}$$

where $y \bmod 1$ denotes the fractional part of a positive real y. Thus R_{i+1} represents the fractional part of the sum of three uniform variates. It is not too difficult to show that $R_{i+1} \sim U(0, 1)$ (see Problem 10).

Since the three generators are maximum period prime modulus, their periods are 30268, 30306, and 30322 respectively. The period of the combined generator is the least common multiple of the individual periods, which is $30268 \times 30306 \times 30324/4 \approx 6.95 \times 10^{12}$. The divisor of 4 arises as the greatest common divisor of the three periods is 2. This is a reliable if rather slow generator. It is used, for example, in Microsoft Excel2003 (http://support.microsoft.com).

2.6 The seed(s) in a random number generator

The seed X_0 of a generator provides the means of reproducing the random number stream when this is required. This is essential when comparing different experimental policies and also when debugging a program. This reproducibility at run-time eliminates the need to store and access a long list of random numbers, which would be both slow and take up substantial memory.

In most cases, *independent* simulation runs (replications, realizations) are required and therefore nonoverlapping sections of the random number sequence should be used. This is most conveniently done by using, as a seed, the last value used in the previous simulation run. If output observations *within* a simulation run are not independent, then this is not sufficient and a buffer of random numbers should be 'spent' before starting a subsequent run. In practice, given the long cycle lengths of many generators, an alternative to these strategies is simply to choose a seed randomly (for example by using the computer's internal clock) and hope that the separate sections of the sequence do not overlap. In that case, results will not be reproducible.

2.7 Problems

1. Generate by hand the complete cycle for the linear recurrences (a) to (f) below. State the observed period and verify that it is in agreement with theory.

   (a) $X_{i+1} = (5X_i + 3) \bmod 16$, $X_0 = 5$;

   (b) $X_{i+1} = (5X_i + 3) \bmod 16$, $X_0 = 7$;

   (c) $X_{i+1} = (7X_i + 3) \bmod 16$, $X_0 = 5$;

   (d) $X_{i+1} = (5X_i + 4) \bmod 16$, $X_0 = 5$;

   (e) $X_{i+1} = (5X_i) \bmod 64$, $X_0 = 3$;

   (f) $X_{i+1} = (5X_i) \bmod 64$, $X_0 = 4$.

2. Modify the procedure 'r1' in Section 2.1.1 to generate numbers in [0,1) using

   (a) $X_{i+1} = (7X_i) \bmod 61$, $X_0 = 1$;

   (b) $X_{i+1} = (49X_i) \bmod 61$, $X_0 = 1$.

   In each case observe the period and verify that it agrees with theory.

3. Verify the correctness of the procedure 'schrage' listed in Section 2.1.1.

4. Consider the maximum period prime modulus generator

$$X_{i+1} = 1000101X_i \bmod (10^{12} - 11)$$

with $X_0 = 53547507752$. Compute by hand $1000101X_0 \bmod (10^{12})$ and $\lfloor 1000101X_0/10^{12} \rfloor$. Hence find X_1 by hand calculation.

5. (a) Consider the multiplicative prime modulus generator

$$X_{i+1} = (aX_i) \bmod m$$

where a is a primitive root of m. Show that over the entire cycle

$$E(X_i) = \frac{m}{2},$$

$$\mathrm{Var}(X_i) = \frac{m(m-2)}{12}.$$

[*Hint*. Use the standard results $\sum_{i=1}^{k} i = k(k+1)/2$ and $\sum_{i=1}^{k} i^2 = k(k+1)(2k+1)/6$.] Put $R_i = X_i/m$ and show that $E(R_i) = \frac{1}{2} \forall m$ and $\mathrm{Var}(R_i) \to 1/12$ as $m \to \infty$.

(b) Let $f(r)$ denote the probability density function of R, a $U(0, 1)$ random variable. Then $f(r) = 1$ when $0 \le r \le 1$ and is zero elsewhere. Let μ and σ denote the mean and standard deviation of R. Show that

$$\mu = \int_0^1 f(r)\,\mathrm{d}r = \frac{1}{2},$$

$$\sigma^2 = \int_0^1 (r-\mu)^2 f(r)\,\mathrm{d}r = \frac{1}{12},$$

thereby verifying that over the *entire* sequence the generator in (a) gives numbers with the correct mean $\forall m$, and with almost the correct variance when m is large.

6. Show that 2 is a primitive root of 13. Hence find all multiplicative linear congruential generators with modulus 13 and period 12.

7. Consider the multiplicative generator

$$X_{i+1} = (aX_i) \bmod 2^b$$

where $a = 5 \bmod 8$. This has a cycle length $m/4 = 2^{b-2}$. The random numbers may be denoted by $R_i = X_i/2^b$. Now consider the mixed full period generator

$$X_{i+1}^* = (aX_i^* + c) \bmod 2^{b-2}$$

where $c = (X_0 \bmod 4)[(a-1)/4]$. Denote the random numbers by $R_i^* = X_i^*/2^{b-2}$. It is shown in Theorem 2.1 that

$$R_i = R_i^* + \frac{X_0 \bmod 4}{2^b}.$$

Verify this result for $b = 5, a = 13$, and $X_0 = 3$ by generating the entire cycles of $\{R_i\}$ and $\{R_i^*\}$.

8. The linear congruential generator obtains X_{i+1} from X_i. A *Fibonacci* generator obtains X_{i+1} from X_i and X_{i-1} in the following way:

$$X_{i+1} = (X_i + X_{i-1}) \bmod m$$

where X_0 and X_1 are specified.

(a) Without writing out a complete sequence, suggest a good upper bound for the period of the generator in terms of m.

(b) Suppose $m = 5$. Only two cycles are possible. Find them and their respective periods and compare with the bound in (a). [Note that an advantage of the Fibonacci generator is that no multiplication is involved – just the addition modulo m. However, the output from such a generator is not too random as all the triples (X_{i-1}, X_i, X_{i+1}) lie on just two planes, $X_{i+1} = X_i + X_{i-1}$ or $X_{i+1} = X_i + X_{i-1} - m$. *Shuffling* the output from such a generator can considerably improve its properties.]

9. (a) Obtain the full cycle of numbers from the generators $X_{i+1} = (7X_i) \bmod 13$ and $Y_{i+1} = (Y_i + 5) \bmod 16$. Using suitable plots, compare the two generators in respect of uniformity of the overlapping pairs $\{(X_i, X_{i+1})\}$ and $\{(Y_i, Y_{i+1})\}$ for $i = 1, 2, \ldots$. What are the periods of the two generators?

(b) Construct a combined $U(0, 1)$ generator from the two generators in (a). The combined generator should have a period greater than either of the two individual generators. When $X_0 = 1$ and $Y_0 = 0$, use this generator to calculate the next two $U(0, 1)$ variates.

10. (a) If U and V are independently distributed random variables that are uniformly distributed in [0,1) show that $(U + V) \bmod 1$ is also $U[0, 1)$. Hence justify the assertion that $R_{i+1} \sim U[0, 1)$ in equation (2.17).

(b) A random number generator in [0, 1) is designed by putting

$$R_n = \left(\frac{X_n}{8} + \frac{Y_n}{7}\right) \bmod 1$$

where $X_0 = 0$, $Y_0 = 1$, $X_{n+1} = (9X_n + 3) \bmod 8$, and $Y_{n+1} = (3Y_n) \bmod 7$ for $n = 0, 1, \ldots$. Calculate $R_0, R_1, \ldots, R_5$. What is the period of the generator, $\{R_n\}$?

11. A $U(0, 1)$ random sequence $\{U_n, n = 0, 1, \ldots\}$ is

0.69	0.79	0.10	0.02	0.43	0.61	0.76	0.66	0.58	....

The pseudo-code below gives a method for shuffling the order of numbers in a sequence, where the first five numbers are entered into the array $\{T\{j\}, j = 0, \ldots, 4\}$.

```
Output T (4) into shuffled sequence
$N := \lfloor 4T(4) \rfloor$
$T(4) := T(N)$
Input next U from unshuffled sequence
$T(N) := U$
```

Obtain the first six numbers in the shuffled sequence.

12. Consider the generator $X_{n+1} = (5X_n + 3) \bmod 16$. Obtain the full cycle starting with $X_0 = 1$. Shuffle the output using the method described in Section 2.3 with $k = 6$. Find the first 20 integers in the shuffled sequence.

13. A frequency test of 10 000 supposed $U(0, 1)$ random numbers produced the following frequency table:

Interval	0–0.1	0.1–0.2	0.2–0.3	0.3–0.4	0.4–0.5	0.5–0.6
Frequency	1023	1104	994	993	1072	930

Interval	0.6–0.7	0.7–0.8	0.8–0.9	0.9–1.0
Frequency	1104	969	961	850

What are your conclusions?