

48 2.2 Lehmer Random Number Generators: Implementation

Recall that one good reason to choose $m = 2^{31} - 1$ as the modulus for a Lehmer random number generator is that virtually all contemporary computer systems support 32-bit 2's complement integer arithmetic and on such systems $2^{31} - 1$ is the largest possible prime. Consistent with that observation, a portable and efficient algorithmic implementation of an $m = 2^{31} - 1$ Lehmer generator that is valid (provided all integers between $-m$ and m can be represented exactly) is developed in this section.

We can use ANSI C for the implementation because 32-bit signed integer arithmetic is supported in a natural way. That is, the ANSI C type `long` is required to be valid for all integers between `LONG_MIN` and `LONG_MAX` inclusive and, although the values are implementation dependent, `LONG_MAX` and `LONG_MIN` are required to be at least $2^{31} - 1$ and at most $-(2^{31} - 1)$ respectively.*

2.2.1 IMPLEMENTATION

If there is no guarantee that integers larger than m can be represented exactly, then the implementation issue of *potential integer overflow* must be addressed. That is, for any full-period Lehmer generator, the product ax can be as large as $a(m-1)$. Therefore, unless values of t as large as $a(m-1)$ can be represented exactly, it is *not* possible to evaluate $g(x) = ax \bmod m$ in the “obvious” way by first computing the intermediate product $t = ax$ and then computing $t \bmod m = t - \lfloor t/m \rfloor m$.

Example 2.2.1 If $(a, m) = (48271, 2^{31} - 1)$ then $a(m-1) \cong 1.47 \times 2^{46}$. Therefore, it would not be possible to implement this (a, m) Lehmer generator in the obvious way without access to a register that is at least 47 bits wide to store the intermediate product t . This is true even though $t \bmod m$ is no more than 31 bits wide. (If $a = 16807$ then a 46 bit register would be required to hold the intermediate product.)

Type Considerations

If we wish to implement a Lehmer generator with $m = 2^{31} - 1$ in ANSI C, and do it in the obvious way, then the type declaration of t will dictate the number of bits available to store the intermediate $t = ax$ product. Correspondingly, if t is (naturally) an *integer* type then the integer division t/m can be used to evaluate $\lfloor t/m \rfloor$; or if t is a *floating point* type then a floating point division t/m followed by a floating-point-to-integer cast can be used to evaluate $\lfloor t/m \rfloor$.

Example 2.2.2 If the variable t is declared to be a `long` and $m = 2^{31} - 1$ then the obvious implementation will be correct only if `LONG_MAX` is $a(m-1)$ or larger. Most contemporary computer systems do not support integers this large and thus for a Lehmer generator with $m = 2^{31} - 1$ the obvious implementation is *not* a viable algorithm option.

* The macros `LONG_MIN` and `LONG_MAX` are defined in the ANSI C library `<limits.h>`. The type `long` is a shorthand representation for the type `long int`. `LONG_LONG` is supported in C99, the new (maybe latest) ANSI/ISO Standard for C for 64-bit integers. This option is only appropriate if the hardware supports 64-bit arithmetic (e.g., Apple, AMD).

Example 2.2.3 If the variable t is declared to be the ANSI C floating point type `double` then the obvious implementation may be correct provided the multiplier a is not too large. That is, `double` is generally consistent with the IEEE 754 64-bit floating point standard which specifies a 53-bit mantissa (including the sign bit) and if $m = 2^{31} - 1$ a mantissa this large allows for t to be much larger than m . So it *may* be possible to implement a $m = 2^{31} - 1$ Lehmer generator with a sufficiently small multiplier in the obvious way by doing the *integer* calculations in *floating point* arithmetic. However, when portability is required and an efficient integer-based implementation is possible, only the unwise would use a floating point implementation instead.

Algorithm Development

Consistent with the previous examples, it is desirable to have an integer-based implementation of Lehmer's algorithm that will port to any system which supports the ANSI C type `long`. This can be done provided no integer calculation produces an intermediate or final result larger than $m = 2^{31} - 1$ in magnitude. With this constraint in mind we must be prepared to do some algorithm development.

If it were possible to factor the modulus as $m = aq$ for some integer q then $g(x)$ could be written as $g(x) = ax \bmod m = a(x \bmod q)$, enabling us to do the mod *before* the multiply and thereby avoid the potential overflow problem. That is, in this case the largest possible value of $ax \bmod m$ would be $a(q-1) = m-a$ and this is less than m . Of course if m is prime no such factorization is possible. It is always possible, however, to "approximately factor" m as $m = aq + r$ with $q = \lfloor m/a \rfloor$ and $r = m \bmod a$. As demonstrated in this section, if the *remainder* r is small relative to the *quotient* q , specifically if $r < q$, then this (q, r) decomposition of m provides the basis for an algorithm to evaluate $g(x)$ in such a way that integer overflow is eliminated.

Example 2.2.4 If $(a, m) = (48271, 2^{31} - 1)$ then the quotient is $q = \lfloor m/a \rfloor = 44488$ and the remainder is $r = m \bmod a = 3399$. Similarly, if $a = 16807$ then $q = 127773$ and $r = 2836$. In both cases $r < q$.

For all $x \in \mathcal{X}_m = \{1, 2, \dots, m-1\}$ define the two functions

$$\gamma(x) = a(x \bmod q) - r\lfloor x/q \rfloor \quad \text{and} \quad \delta(x) = \lfloor x/q \rfloor - \lfloor ax/m \rfloor.$$

Then, for any $x \in \mathcal{X}_m$

$$\begin{aligned} g(x) &= ax \bmod m = ax - m\lfloor ax/m \rfloor \\ &= ax - m\lfloor x/q \rfloor + m\lfloor x/q \rfloor - m\lfloor ax/m \rfloor \\ &= ax - (aq + r)\lfloor x/q \rfloor + m\delta(x) \\ &= a(x - q\lfloor x/q \rfloor) - r\lfloor x/q \rfloor + m\delta(x) \\ &= a(x \bmod q) - r\lfloor x/q \rfloor + m\delta(x) = \gamma(x) + m\delta(x). \end{aligned}$$

An efficient, portable implementation of a Lehmer random number generator is based upon this alternate representation of $g(x)$ and the following theorem.

Theorem 2.2.1 If $m = aq + r$ is prime, $r < q$, and $x \in \mathcal{X}_m$, then $\delta(x) = \lfloor x/q \rfloor - \lfloor ax/m \rfloor$ is either 0 or 1. Moreover, with $\gamma(x) = a(x \bmod q) - r\lfloor x/q \rfloor$

- $\delta(x) = 0$ if and only if $\gamma(x) \in \mathcal{X}_m$;
- $\delta(x) = 1$ if and only if $-\gamma(x) \in \mathcal{X}_m$.

Proof First observe that if u and v are real numbers with $0 < u - v < 1$ then the integer difference $\lfloor u \rfloor - \lfloor v \rfloor$ is either 0 or 1. Therefore, because $\delta(x) = \lfloor x/q \rfloor - \lfloor ax/m \rfloor$, the first part of the theorem is true if we can show that

$$0 < \frac{x}{q} - \frac{ax}{m} < 1.$$

Rewriting the center of the inequality as

$$\frac{x}{q} - \frac{ax}{m} = x \left(\frac{1}{q} - \frac{a}{m} \right) = x \left(\frac{m - aq}{mq} \right) = \frac{xr}{mq}$$

and because $r < q$

$$0 < \frac{xr}{mq} < \frac{x}{m} \leq \frac{m-1}{m} < 1$$

which establishes that if $x \in \mathcal{X}_m$ then $\delta(x)$ is either 0 or 1. To prove the second part of this theorem recall that if $x \in \mathcal{X}_m$ then $g(x) = \gamma(x) + m\delta(x)$ is in \mathcal{X}_m . Therefore, if $\delta(x) = 0$ then $\gamma(x) = g(x)$ is in \mathcal{X}_m ; conversely if $\gamma(x) \in \mathcal{X}_m$ then $\delta(x)$ cannot be 1 for if it were then $g(x) = \gamma(x) + m$ would be $m + 1$ or larger which contradicts $g(x) \in \mathcal{X}_m$. Similarly, if $\delta(x) = 1$ then $-\gamma(x) = m - g(x)$ is in \mathcal{X}_m ; conversely if $-\gamma(x) \in \mathcal{X}_m$ then $\delta(x)$ cannot be 0 for if it were then $g(x) = \gamma(x)$ would not be in \mathcal{X}_m .

Algorithm

The key to avoiding overflow in the evaluation of $g(x)$ is that the calculation with the potential to cause overflow, the ax product, is “trapped” in $\delta(x)$. This is important because, from Theorem 2.2.1, the value of $\delta(x)$ can be deduced from the value of $\gamma(x)$ and it can be shown that $\gamma(x)$ can be computed without overflow — see Exercise 2.2.5. This comment and Theorem 2.2.1 can be summarized with the following algorithm.

Algorithm 2.2.1 If $m = aq + r$ is prime, $r < q$, and $x \in \mathcal{X}_m$, then $g(x) = ax \bmod m$ can be evaluated as follows without producing any intermediate or final values larger than $m - 1$ in magnitude.

```

t = a * (x % q) - r * (x / q);           /* t = γ(x) */
if (t > 0)
    return (t);                          /* δ(x) = 0 */
else
    return (t + m);                       /* δ(x) = 1 */

```

Modulus Compatibility

Definition 2.2.1 The multiplier a is *modulus-compatible* with the prime modulus m if and only if the remainder $r = m \bmod a$ is less than the quotient $q = \lfloor m/a \rfloor$.

If a is modulus-compatible with m then Algorithm 2.2.1 can be used as the basis for an implementation of a Lehmer random number generator. In particular, if the multiplier is modulus-compatible with $m = 2^{31} - 1$ then Algorithm 2.2.1 can be used to implement the corresponding Lehmer random number generator in such a way that it will port to any system that supports 32-bit integer arithmetic. From Example 2.2.4, for example, the full-period multiplier $a = 48271$ is modulus-compatible with $m = 2^{31} - 1$.

In general, there are no modulus-compatible multipliers beyond $(m - 1)/2$. Moreover, as the following example illustrates, modulus-compatible multipliers are much more densely distributed on the low end of the $1 \leq a \leq (m - 1)/2$ scale.

Example 2.2.5 The (tiny) modulus $m = 401$ is prime. Figure 2.2.1 illustrates, on the first line, the 38 associated modulus-compatible multipliers. On the second line are the 160 full-period multipliers and the third line illustrates the ten multipliers (3, 6, 12, 13, 15, 17, 19, 21, 23, and 66) that are *both* modulus-compatible and full-period.

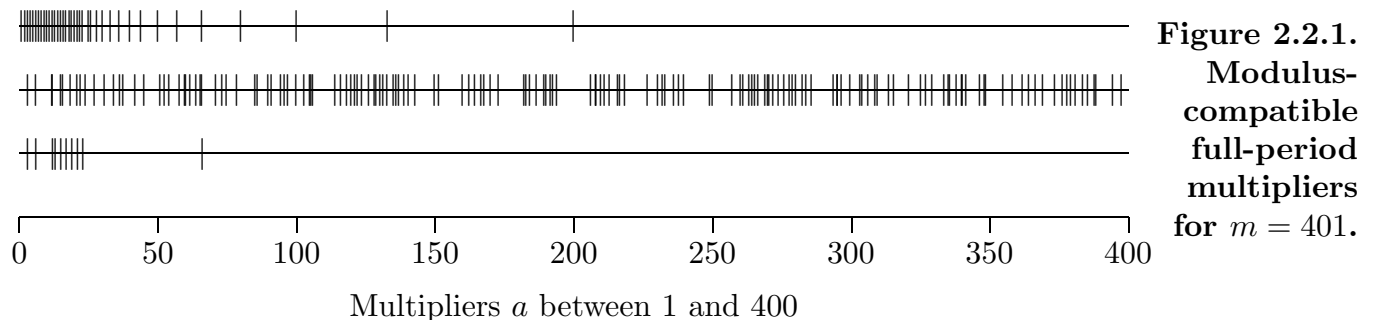


Figure 2.2.1.
Modulus-compatible full-period multipliers for $m = 401$.

If you have a very high resolution graphics device, a very long piece of paper, a magnifying glass, and a few hours of CPU time, you are encouraged to make the corresponding figure for $m = 2^{31} - 1$. (For this modulus there are 92679 modulus-compatible multipliers. Of these, 23093 are also full-period — see Example 2.2.6.)

Modulus-compatibility is closely related to the smallness of the multiplier. In particular, a is defined to be “small” if and only if $a^2 < m$. If a is small then a is modulus-compatible with m — see Exercise 2.2.4. From this result it follows, for example, that all the multipliers between 1 and 46340 inclusive are modulus-compatible with $m = 2^{31} - 1$. Thus smallness is sufficient to guarantee modulus-compatibility. Smallness is not a necessary condition, however. In particular, $a = 48271$ is modulus-compatible with $m = 2^{31} - 1$, but it is not small. Similarly, of the 38 modulus-compatible multipliers for $m = 401$ in Example 2.2.5, only about half (those between 1 and 20) satisfy the $a^2 < m$ small multiplier test.

As illustrated in Example 2.2.5, for a given prime modulus there are relatively few associated full-period, modulus-compatible multipliers. A mechanical way to find one of these is to start with the small (and thus modulus-compatible) multipliers $a = 2$, $a = 3$, etc.; until the first full-period multiplier is found — see Algorithm 2.1.1. Given that we have found one full-period, modulus-compatible multiplier, the following $O(m)$ algorithm can be used to generate *all* the others. This algorithm, an extension of Algorithm 2.1.2, presumes the availability of a function `gcd` that returns the greatest common divisor of two positive integers (see Appendix B).

Algorithm 2.2.2 Given the prime modulus m and any associated full-period, modulus-compatible multiplier a the following algorithm generates all the full-period, modulus-compatible multipliers relative to m .

```

i = 1;
x = a;
while (x != 1) {
    if ((m % x < m / x) and (gcd(i, m - 1) == 1))
        /* x is a full-period modulus-compatible multiplier */
        i++;
    x = g(x);      /* use Algorithm 2.2.1 to evaluate g(x) = ax mod m */
}

```

Example 2.2.6 Using Algorithm 2.2.2 with $m = 2^{31} - 1 = 2147483647$, $a = 7$ and a lot of CPU cycles we find that there are a total of 23093 associated full-period, modulus-compatible multipliers, the first few of which are

$$\begin{aligned}
 7^1 \bmod 2147483647 &= 7 \\
 7^5 \bmod 2147483647 &= 16807 \\
 7^{113039} \bmod 2147483647 &= 41214 \\
 7^{188509} \bmod 2147483647 &= 25697 \\
 7^{536035} \bmod 2147483647 &= 63295.
 \end{aligned}$$

Of these, the multiplier $a = 16807$ deserves special mention. It was first suggested by Lewis, Goodman, and Miller (1969), largely because it was easy to prove (as we have done) that it is a full-period multiplier. Since then it has become something of a “minimal” standard (Park and Miller, 1988).

In retrospect $a = 16807$ was such an obvious choice that it seems unlikely to be the best possible full-period multiplier relative to $m = 2^{31} - 1$ and, indeed, subsequent testing for randomness has verified that, at least in theory, other full-period multipliers generate (slightly) more random sequences. Although several decades of generally favorable user experience with $a = 16807$ is not easily ignored, we will use $a = 48271$ instead.

Randomness

For a given (prime) modulus m , from among all the full-period, modulus-compatible multipliers relative to m we would like to choose the one that generates the “most random” sequence. As suggested in Example 2.1.1, however, there is no simple and universal definition of randomness and so it should come as no surprise to find that there is less than complete agreement on what this metric should be.

Example 2.2.7 To the extent that there is agreement on a randomness metric, it is based on a fundamental geometric characteristic of all Lehmer generators, first established by Marsaglia (1968), that “random numbers fall mainly in the planes.” That is, in 2-space for example, the points

$$(x_0, x_1), (x_1, x_2), (x_2, x_3), \dots$$

all fall on a finite, and possibly small, number of parallel lines to form a *lattice* structure as illustrated in Figure 2.2.2 for $(a, m) = (23, 401)$ and $(66, 401)$. If a simulation model of a nuclear power plant, for example, required two consecutive small random numbers (e.g., both smaller than 0.08) for a particular rare event (e.g., melt down), this would *never* occur for the $(66, 401)$ generator due to the void in the southwest corner of the graph. [Using the axiomatic approach to probability, the exact probability is $(0.08)(0.08) = 0.0064$.]

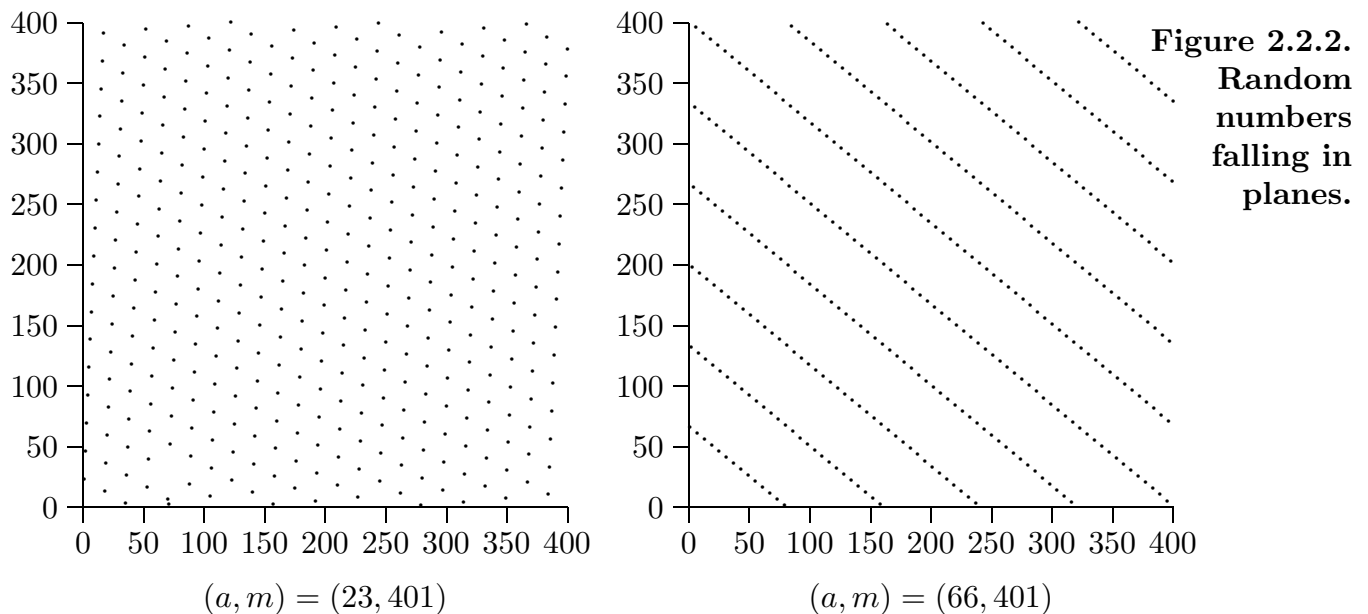


Figure 2.2.2.
Random numbers falling in planes.

In general, for any integer $k \geq 2$ the points

$$(x_0, x_1, \dots, x_{k-1}), (x_1, x_2, \dots, x_k), (x_2, x_3, \dots, x_{k+1}), \dots$$

form a lattice structure. The best Lehmer generator randomness metrics are based on analyzing (numerically, not visually) the *uniformity* of this lattice structure in k -space for small values of k . Knuth (1998) gives one popular randomness metric of this type known as the *spectral test*. Consistent with these metrics, for $m = 401$ the multiplier $a = 23$ would be a much better full-period, modulus-compatible multiplier than $a = 66$.

ANSI C Implementation

The kind of theoretical testing for randomness illustrated in Example 2.2.7 has been done for all of the 23093 full-period, modulus-compatible multipliers relative to $m = 2^{31} - 1$. Of these, the winner is $a = 48271$ with $q = \lfloor m/a \rfloor = 44488$ and $r = m \bmod a = 3399$.*

Example 2.2.8 A Lehmer random number generator with $(a, m) = (48271, 2^{31} - 1)$ can be implemented correctly, efficiently and portably in ANSI C as follows

```
double Random(void)
{
    const long A      = 48271;           /* multiplier */
    const long M      = 2147483647;     /* modulus    */
    const long Q      = M / A;         /* quotient   */
    const long R      = M % A;         /* remainder  */
    static long state = 1;
        long t      = A * (state % Q) - R * (state / Q);
    if (t > 0)
        state = t;
    else
        state = t + M;
    return ((double) state / M);
}
```

With minor implementation-dependent modifications, the random number generator in Example 2.2.8 is the basis for all the simulated stochastic results presented in this book. There are three important points relative to this particular implementation.

- The static variable `state` is used to hold the current *state* of the random number generator, initialized to 1. There is nothing magic about 1 as an initial state (seed); any value between 1 and 2 147 483 646 could have been used.
- Because `state` is a static variable, the state of the generator will be retained between successive calls to `Random`, as must be the case for the generator to operate properly. Moreover, because the scope of `state` is local to the function, the state of the generator is protected — it cannot be changed in any way other than by a call to `Random`.
- If the implementation is correct, tests for randomness are redundant; if the implementation is incorrect, it should be discarded. A standard way of testing for a correct implementation is based on the fact that if the initial value of `state` is 1, then after 10 000 calls to `Random` the value of `state` should be 399 268 537.

* In addition to “in theory” testing, the randomness associated with this (a, m) pair has also been subjected to a significant amount of “in practice” empirical testing — see Section 10.1.

Example 2.2.9 As a potential alternative to the generator in Example 2.2.8, the random number generator in the ANSI C library `<stdlib.h>` is the function `rand`. The intent of this function is to simulate drawing at random from the set $\{0, 1, 2, \dots, m - 1\}$ with m required to be at least 2^{15} . That is, `rand` returns an `int` between 0 and `RAND_MAX` inclusive where the macro constant `RAND_MAX` (defined in the same library) is required to be at least $2^{15} - 1 = 32767$. To convert the integer value returned by `rand` to a floating point number between 0.0 and 1.0 (consistent with Definition 2.1.1) it is conventional to use an assignment like

```
u = (double) rand() / RAND_MAX;
```

Note, however, that the ANSI C standard does *not* specify the details of the algorithm on which this generator is based. Indeed, the standard does not even require the output to be random! For scientific applications it is generally a good idea to avoid using `rand`, as indicated in Section 13.15 of Summit (1995).

Random Number Generation Library

The random number generation library used in this course is based upon the implementation considerations developed in this section. This library is defined by the header file `"rng.h"` and is recommended as a replacement for the standard ANSI C library functions `rand` and `srand`, particularly in simulation applications where the statistical goodness of the random number generator is important. The library provides the following capabilities.

- `double Random(void)` — This is the Lehmer random number generator in Example 2.2.8. We recommended it as a replacement for the standard ANSI C library function `rand`.
- `void PutSeed(long seed)` — This function can be used to initialize or reset the current state of the random number generator. We recommended it as a replacement for the standard ANSI C library function `srand`. If `seed` is positive then that value becomes the current state of the generator. If `seed` is 0 then the user is prompted to set the state of the generator interactively via keyboard input. If `seed` is negative then the state of the generator is set by the system clock.*
- `void GetSeed(long *seed)` — This function can be used to get the current state of the random number generator.*
- `void TestRandom(void)` — This function can be used to test for a correct implementation of the library.

Although we recommend the use of the multiplier 48271, and that is the value used in the library `rng`, as discussed in Example 2.2.6 the 16807 multiplier is something of a minimal standard. Accordingly, the library is designed so that it is easy to use 16807 as an alternative to 48271.

* See Section 2.3 for more discussion about the use of `PutSeed` and `GetSeed`.

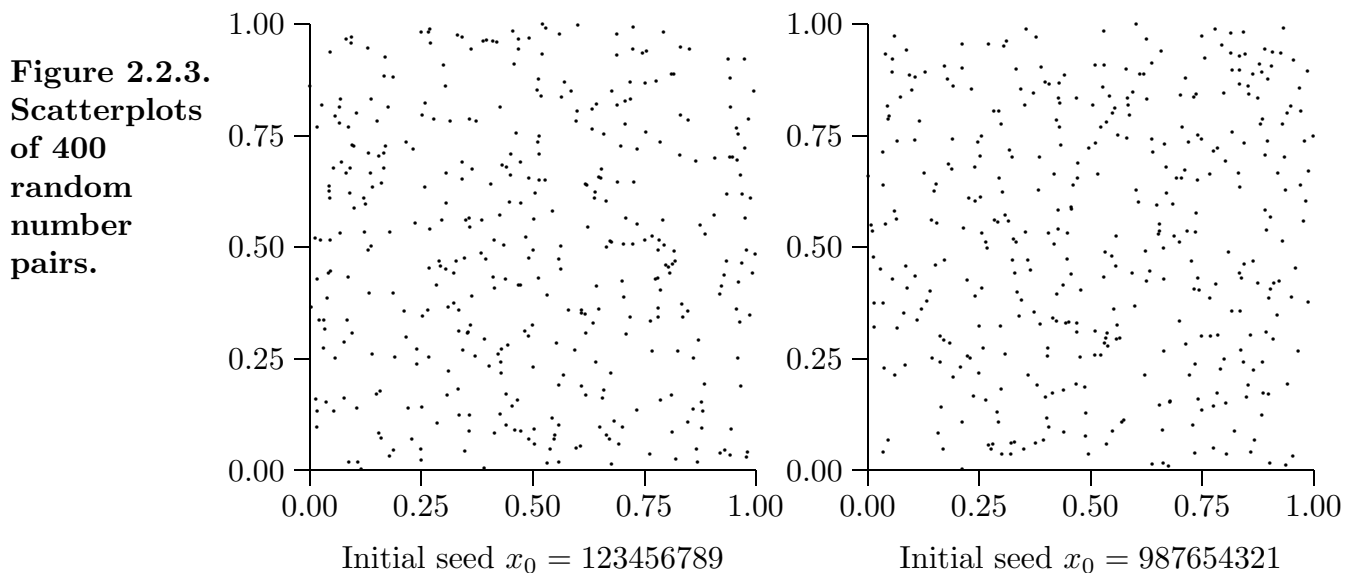
Example 2.2.10 As an example of the use of `Random` and `PutSeed`, this algorithm was used to create the two scatterplots illustrated in Figure 2.2.3 for two different values of the initial seed.

```

seed = 123456789;                               /* or 987654321 */
PutSeed(seed);
x0 = Random();
for (i = 0; i < 400; i++) {
    xi+1 = Random();
    Plot(xi, xi+1);                          /* a generic graphics function */
}

```

Unlike the lattice structure so obvious in Example 2.2.7, the (x_i, x_{i+1}) pairs in this case appear to be random with no lattice structure evident, as desired. (For a more direct comparison all the integer-valued coordinates in Example 2.2.7 would need to be normalized to 1.0 via division by $m = 401$. That would be a purely cosmetic change, however.)



Consistent with the discussion in Example 2.2.7, it should be mentioned that in a sense the appearance of randomness in Example 2.2.10 is an illusion. That is, if *all* of the possible pairs of (x_i, x_{i+1}) points were generated (there are $m - 1 = 2^{31} - 2$ of these) and if it were somehow possible to plot all of these pairs of points at a fine enough scale to avoid blackening the page, then like the figures in Example 2.2.7 a micro-scale lattice structure would be evident.* This observation is one of the reasons why we distinguish between *ideal* and *good* random number generators (see Definition 2.1.1).

* Contemplate what size graphics device and associated dpi (dots per inch) resolution would be required to actually do this.

Example 2.2.11 Plotting consecutive, overlapping random number pairs (x_i, x_{i+1}) from a full-period Lehmer generator with $m = 2^{31} - 1$ would blacken the unit square, obscuring the lattice structure. The fact that any tiny square contained in the unit square will exhibit approximately the same appearance is exploited in the algorithm below, where *all* of the random numbers are generated by `Random()`, but only those that fall in the square with opposite corners $(0, 0)$ and $(0.001, 0.001)$ are plotted. One would expect that approximately $(0.001)(0.001)(2^{31} - 2) \cong 2147$ of the points would fall in the tiny square.

```
seed = 123456789;
PutSeed(seed);
x0 = Random();
for (i = 0; i < 2147483646; i++) {
    x_{i+1} = Random();
    if ((x_i < 0.001) and (x_{i+1} < 0.001)) Plot(x_i, x_{i+1});
}
```

The results of the implementation of the algorithm are displayed in Figure 2.2.4 for the multipliers $a = 16807$ (on the left) and $a = 48271$ (on the right). The random numbers produced by $a = 16807$ fall in just 17 nearly-vertical parallel lines, whereas the random numbers produced by $a = 48271$ fall in 47 parallel lines. These scatterplots provide further evidence for our choice of $a = 48271$ over $a = 16807$ in the random number generator provided in the library `rng`.

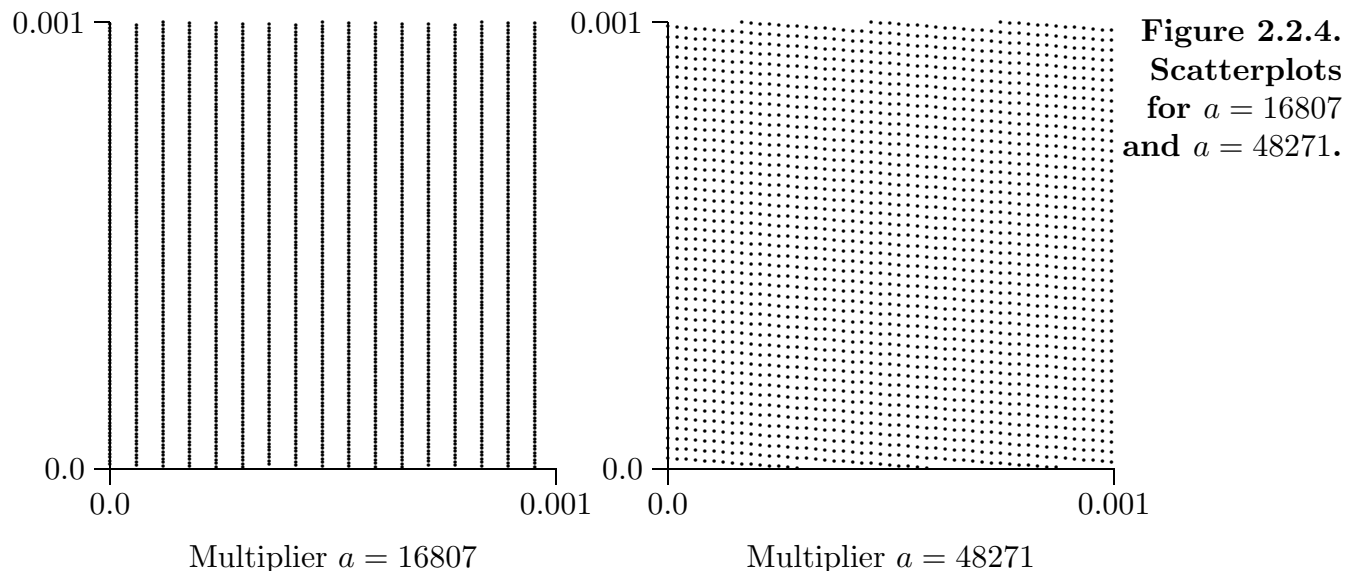


Figure 2.2.4.
Scatterplots
for $a = 16807$
and $a = 48271$.

We have shown that as we zoom in on the unit square the (unwanted) lattice structure of the (x_i, x_{i+1}) pairs produced by our *good* random number generator becomes more apparent. It is appropriate at this juncture to contemplate what would happen with an *ideal* random number generator. An ideal generator exhibits no lattice behavior as you zoom in on increasingly tiny squares. Like a fractal or a scale-free network, all scatterplots will look like those in Figure 2.2.3, assuming more points are generated as you zoom in.

Other Multipliers

Recall that for the modulus $m = 2^{31} - 1 = 2\,147\,483\,647$, there are 534 600 000 multipliers a that result in a full-period random number generator. Of these, 23 093 are modulus-compatible. Although the modulus-compatible, full-period multiplier $a = 48\,271$ is used in library `rng`, are there other multipliers that are recommended? One rather tedious way of approaching this question is to execute the statistical tests for random number generators outlined in Section 10.1 on all of the generators. A second, more efficient approach is to consult the experts who perform research in this area. Fishman (2001, pages 428–445) notes that $a = 16\,807$, $a = 630\,360\,016$, and $a = 742\,938\,285$ are employed by the discrete-event simulation languages SIMAN, SIMSCRIPT II.5, and GPSS/H, respectively, ordered here by improving lattice structure. In addition to $a = 742\,938\,285$, he also suggests four other multipliers with similar performance in terms of parallel hyperplanes in 2, 3, 4, 5, and 6 dimensions: $a = 950\,706\,376$, $a = 1\,226\,874\,159$, $a = 62\,089\,911$, and $a = 1\,343\,714\,438$. Determining whether these multipliers are modulus-compatible is left as an exercise.

Good Generator — Non-representative Subsequences

In any simulation run, only a subset of the random numbers produced by a generator are typically used. What if, for example, only 20 random numbers were needed for a simulation. If you had the *extreme* misfortune of choosing the seed $x_0 = 109\,869\,724$ with the generator in the library `rng`, the resulting 20 random numbers (to only 0.*xx* precision) are:

0.64 0.72 0.77 0.93 0.82 0.88 0.67 0.76 0.84 0.84
 0.74 0.76 0.80 0.75 0.63 0.94 0.86 0.63 0.78 0.67.

Is there something wrong with this generator to have produced 20 consecutive random numbers exceeding 0.62? Certainly not. It will only occur once in a blue moon, but this particular seed resulted in this rather unique sequence of random numbers.* This sequence of 20 consecutive random numbers may initially seem analogous to an *outlier* from statistics. Statisticians sometimes discard outliers as being unrepresentative, but it is *never* appropriate to do so in simulation. The random number generator will have a few sequences of unusually high random numbers as well as a few sequences of unusually low random numbers, as it should. Analogously, if we were to perform the experiment of tossing a fair coin 20 times for a large number of replications, there should be a few rare cases when all 20 tosses come up heads (or tails).

This string of 20 large random numbers highlights the importance of *replicating* a simulation many times so as to average out these unusual cases. Replication will be considered in detail in Chapter 8.

* The probability of 20 consecutive random numbers exceeding 0.62 using the axiomatic approach to probability is $(0.38)^{20} \cong 4 \cdot 10^{-9}$.

The Curse of Fast CPUs

When discrete-event simulation was in its infancy, the time required to cycle through a full-period Lehmer generator with modulus $m = 2^{31} - 1 = 2\,147\,483\,647$ was measured in days. This quickly shrank to hours, and now only takes a few minutes on a desktop machine. Before long, the time to complete a cycle of such a generator will be measured in seconds. The problem of *cycling*, that is, reusing the same random numbers in a single simulation, must be avoided because of the resultant dependency in the simulation output.

Extending the period length of Lehmer generators has been an active research topic within the simulation community for many years. One simple solution is to run all simulations on 64-bit machines. Since the largest prime number less than $2^{63} - 1$ is $2^{63} - 25 = 9\,223\,372\,036\,854\,775\,783$, the period is lengthened from about $2.14 \cdot 10^9$ to $9.22 \cdot 10^{18}$. In this case, cycle times are measured in years. Obviously, more machines supporting 64-bit integer arithmetic is beneficial to the simulation community. Since it is too early to know if 64-bit machines will become the norm, other portable techniques on 32-bit machines for achieving longer periods have been developed.

One technique for achieving a longer period is to use a *multiple recursive generator*, where our usual $x_{i+1} = ax_i \bmod m$ is replaced by

$$x_{i+1} = (a_1x_i + a_2x_{i-1} + \cdots + a_qx_{i-q}) \bmod m.$$

These generators can produce periods as long as $m^q - 1$ if the parameters are chosen properly. A second technique for extending the period is to use a *composite generator*, where several Lehmer generators can be combined in a manner to extend the period and improve statistical behavior. A third technique is to use a *Tausworthe*, or *shift-register generator*, where the modulo function is applied to bits, rather than large integers. The interested reader should consult Chapter 6 of Bratley, Fox, and Schrage (1987), Chapter 5 of Lewis and Orav (1989), Chapter 7 of Law and Kelton (2000), Chapter 9 of Fishman (2001), Gentle (2003), and L'Ecuyer, Simard, Chen, and Kelton (2002) for more details.

Any of these generators that have been well-tested by multiple authors yield two benefits: longer periods and better statistical behavior (e.g., fewer hyperplanes). The cost is always the same: increased CPU time.

2.2.2 EXERCISES

Exercise 2.2.1 Prove that if u and v are real numbers with $0 < u - v < 1$ then the integer difference $\lfloor u \rfloor - \lfloor v \rfloor$ is either 0 or 1.

Exercise 2.2.2^a If $g(\cdot)$ is a Lehmer generator (full period or not) then there must exist an integer $x \in \mathcal{X}_m$ such that $g(x) = 1$. (a) Why? (b) Use the $m = aq + r$ decomposition of m to derive a $O(a)$ algorithm that will solve for this x . (c) If $a = 48271$ and $m = 2^{31} - 1$ then what is x ? (d) Same question if $a = 16807$.

Exercise 2.2.3^a Derive a $O(\log(a))$ algorithm to solve Exercise 2.2.2. *Hint:* a and m are relatively prime.

Exercise 2.2.4 Prove that if a, m are positive integers and if a is “small” in the sense that $a^2 < m$, then $r < q$ where $r = m \bmod a$ and $q = \lfloor m/a \rfloor$.

Exercise 2.2.5 Write $\gamma(x) = \alpha(x) - \beta(x)$ where $\alpha(x) = a(x \bmod q)$ and $\beta(x) = r \lfloor x/q \rfloor$ with $m = aq + r$ and $r = m \bmod a$. Prove that if $r < q$ then for all $x \in \mathcal{X}_m$ both $\alpha(x)$ and $\beta(x)$ are in $\{0, 1, 2, \dots, m-1\}$.

Exercise 2.2.6 Is Algorithm 2.2.1 valid if m is not prime? If not, how should it be modified?

Exercise 2.2.7 (a) Implement a correct version of `Random` using floating point arithmetic and do a timing study. (b) Comment.

Exercise 2.2.8 Prove that if a, x, q are positive integers then $ax \bmod aq = a(x \bmod q)$.

Exercise 2.2.9 You have been hired as a consultant by *XYZ Inc* to assess the market potential of a relatively inexpensive *hardware* random number generator they may develop for high-speed scientific computing applications. List all the technical reasons you can think of to convince them this is a bad idea.

Exercise 2.2.10 There are exactly 400 points in each of the figures in Example 2.2.7. (a) Why? (b) How many points would there be if a were not a full-period multiplier?

Exercise 2.2.11 Let m be the largest prime modulus less than or equal to $2^{15} - 1$ (see Exercise 2.1.6). (a) Compute all the corresponding modulus-compatible full-period multipliers. (b) Comment on how this result relates to random number generation on systems that support 16-bit integer arithmetic only.

Exercise 2.2.12^a (a) Prove that if m is prime with $m \bmod 4 = 1$ then a is a full-period multiplier if and only if $m - a$ is also a full-period multiplier. (b) What if $m \bmod 4 = 3$?

Exercise 2.2.13^a If $m = 2^{31} - 1$ compute the $x \in \mathcal{X}_m$ for which $7^x \bmod m = 48271$.

Exercise 2.2.14 The lines on the scatterplot in Figure 2.2.4 associated with the multiplier $a = 16807$ appear to be vertical. Argue that the lines must *not* be vertical based on the fact that $(a, m) = (16807, 2^{31} - 1)$ is a full-period generator.

Exercise 2.2.15 Determine whether the multipliers associated with $m = 2^{31} - 1$ given by Fishman (2001): $a = 630\,360\,016$, $a = 742\,938\,285$, $a = 950\,706\,376$, $a = 1\,226\,874\,159$, $a = 62\,089\,911$, and $a = 1\,343\,714\,438$ are modulus-compatible.