COMPUTING PRACTICES

Edgar H. Sibley
Panel Editor

Practical and theoretical issues are presented concerning the design, implementation, and use of a good, minimal standard random number generator that will port to virtually all systems.

RANDOM NUMBER GENERATORS: GOOD ONES ARE HARD TO FIND

STEPHEN K. PARK AND KEITH W. MILLER

An important utility that digital computer systems should provide is the ability to generate random numbers. Certainly this is true in scientific computing where many years of experience has demonstrated the importance of access to a good random number generator. And in a wider sense, largely due to the encyclopedic efforts of Donald Knuth [18], there is now a realization that random number generation is a concept of fundamental importance in many different areas of computer science. Despite that, the widespread adoption of good, portable, *industry standard* software for random number generation has proven to be an elusive goal. Many generators have been written, most of them have demonstrably non-random characteristics, and some are embarrassingly bad. In fact, the current state of random number generation software is accurately described by Knuth [18, p. 176] who advises "... look at the subroutine library of each computer installation in your organization, and replace the random number generators by good ones. Try to avoid being too shocked at what you find."

Knuth's advice applies equally well to most recently published computer science textbooks, particularly those written for the undergraduate market. Indeed, during the preparation of this article we reviewed more than 50 computer science textbooks that contained software for at least one random number generator. Most of these generators are unsatisfactory.

This article was motivated by practical software considerations developed over a period of several years while teaching a graduate level course in simulation. Students taking this course work on a variety of systems and their choices typically run the gamut from personal computers to mainframes. With Knuth's advice in mind, one important objective of this course is for all students to write and use implementations of a good, minimal standard random number generator that will port to *all* systems. For reasons discussed later, this minimal standard is a multiplicative linear congruential generator [18, p. 10] with multiplier 16807 and prime modulus $2^{31} - 1$. As it turns out, porting this random number generator (or any other for that matter) to a wide variety of systems is not as easy as it may seem. The issues involved are discussed later in this article.

The body of this article is organized into four sections. In the first, we present the rationale for our choice of a minimal standard generator. We believe that this is the generator that should always be used—unless one has access to a random number generator *known* to be better. In the second section we demonstrate how to implement the minimal standard in a high-level language on a variety of systems. The third section presents theoretical considerations and implementation details in support of the discussion in the previous sections. Finally, in the last section, we present selected examples of unsatisfactory generators that have either appeared in recently published (post-1980) computer science textbooks or are currently supplied by popular programming environments.

MINIMAL STANDARD

To the non-specialist, the construction of a random number generator may appear to be the kind of thing that any good programmer can do easily. Over the years many programmers have unwittingly demonstrated that it is all too easy to 'hack' a procedure that will produce a strange looking, apparently unpredictable sequence of numbers. It is fundamentally more difficult, however, to write quality software which produces what is really desired—a virtually infinite sequence of *statistically independent* random numbers, *uniformly* distributed between 0 and 1. This is a key point: strange and unpredictable is not necessarily random.

In retrospect it is evident that a generally satisfactory algorithm for random number generation was proposed by D. H. Lehmer 36 years ago [26]. This parametric multiplicative linear congruential algorithm has withstood the test of time. It can be implemented efficiently [27, 31, 37, 41], numerous empirical tests of the randomness of its output have been published [8, 15, 27, 28, 37], and its important theoretical properties have been analyzed [9, 14, 18, 30]. The conclusion to be drawn from all this research is now clear. Although Lehmer's algorithm has some statistical defects, *if* the algorithm parameters are chosen properly and *if* the software implementation of the algorithm is correct, the resulting generator can produce a virtually infinite sequence of numbers that will satisfy almost any statistical test of randomness. In other words, with properly chosen parameters, Lehmer's algorithm, correctly implemented, represents a good *minimal standard* generator against which all other random number generators can—and should—be judged.

Lehmer's algorithm represents a good example of the elegance of simplicity. Specifically, the algorithm involves nothing more than the judicious choice of two fixed integer parameters

(i) *modulus*: m—a large *prime* integer

(ii) *multiplier*: a—an integer in the range $2, 3, \ldots, m-1$

and the subsequent generation of the integer sequence $z_1, z_2, z_3 \ldots$ via the iterative equation

(iii) $z_{n+1} = f(z_n) \quad \text{for} \quad n = 1, 2, \ldots$

where the *generating function* $f(\cdot)$ is defined for all z in $1, 2, \ldots, m-1$ as

(iv) $f(z) = az \bmod m$.

The sequence of z's must be initialized by choosing an *initial seed* z_1 from $1, 2, \ldots, m-1$. And, as an additional step, the sequence is conventionally normalized to the unit interval via division by the modulus to produce the real sequence $u_1, u_2, u_3, \ldots$ where

(v) $u_n = z_n/m \quad \text{for} \quad n = 1, 2, \ldots$

A random number generator based on this algorithm is known formally as a prime modulus multiplicative linear congruential generator (PMMLCG) [22]. We prefer the less formal term Lehmer generator.

Several things should be noted. First, because m is prime, $f(z) \neq 0$ for all z in $1, 2, \ldots, m-1$. This is important because it prevents the sequence of z's from collapsing to zero. Second, the values $u = 0$ and $u = 1$ are impossible. Instead, the smallest and largest possible values of u are $1/m$ and $1 - 1/m$ respectively. Third, the normalization by m does not affect the fundamental issue of whether or not the sequence of u's *appears* to be random. That is, the issue of randomness can be completely resolved by studying the integer sequence of z's.

The genius of Lehmer's algorithm is that if the multiplier and prime modulus are properly chosen, the resulting sequence of z's will be statistically indistinguishable from a sequence drawn at random (albeit without replacement) from the set $1, 2, \ldots, m-1$. Indeed, it is *only* in the sense of simulating this random draw that the algorithm is random—there is actually nothing random about Lehmer's algorithm (except possibly the choice of the initial seed). For this reason Lehmer generators are sometimes labeled *pseudo*random.

For instance, consider an example defined by $f(z) = 6z \bmod 13$. If the initial seed is $z_1 = 1$ then the resulting sequence of z's is

$$\ldots 1, 6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1 \ldots \qquad (1)$$

where, as the ellipses indicate, the sequence is actually periodic because it begins to cycle (with a full period of length $m - 1 = 12$) when the initial seed reappears. The point is that the first 12 terms of this sequence (or indeed any 12 consecutive terms) appear to have been drawn at random, without replacement, from the set $1, 2, \ldots, 12$. Also, because $f(z) = 6z \bmod 13$ is a *full period* generating function, *any* initial seed between 1 and 12 could have been chosen without affecting the apparent randomness of the sequence. For example, if the initial seed is 2, the resulting sequence is

$$\ldots 2, 12, 7, 3, 5, 4, 11, 1, 6, 10, 8, 9, 2 \ldots \qquad (2)$$

which is nothing more than a circular shift of sequence (1). In general *all* full period Lehmer generators behave just like this example—they produce a fixed virtual circular list defined by a permutation of the integers $1, 2, \ldots, m-1$. The initial seed provides an initial list element, all other elements are then drawn in sequence.

This example also illustrates the importance of a proper choice of multiplier. Specifically, if the multiplier is changed from 6 to 7, the resulting full period sequence generated by $f(z) = 7z \bmod 13$ is

$$\ldots 1, 7, 10, 5, 9, 11, 12, 6, 3, 8, 4, 2, 1, \ldots \qquad (3)$$

In a sense, randomness, like beauty, is in the eye of the beholder. Because of the patterns evident in the second half of this sequence, however, most people would consider (3) to be *less random* than (1). Thus, even though (6z mod 13) and (7z mod 13) are both full generating functions, the former is a better choice as it produces a *more random* output.

Continuing with this example, if the multiplier is changed to a 5, the resulting sequence generated by $f(z) = 5z \bmod 13$ does not even have a full period. Specifically, either

$$\ldots 1, 5, 12, 8, 1, \ldots \quad \text{or} \quad \ldots 2, 10, 11, 3, 2, \ldots \quad \text{or} \quad \ldots 4, 7, 9, 6, 4, \ldots \tag{4}$$

is generated depending on the choice of initial seed. This latter type of small-period behavior is clearly undesirable—and avoidable. It is known that for *any* prime modulus ($m \geq 3$) a significant percentage of the $m - 2$ possible choices for a will yield a full period generating function. (Specifically, for $m = 13$ the full period multipliers are $a = 2, 6, 7, 11$ and for $m = 2^{31} - 1$, $a = 16807$ is just one of more than 534 *million* full period multipliers [9].) Thus there is no good reason to use a Lehmer generator without a full period.

The previous example illustrates two of the three central issues that must be resolved when creating a Lehmer generator—full period periodicity and randomness. The third central issue is implementation, that is, guaranteeing that $f(z) = az \bmod m$ will be evaluated efficiently and correctly for *all* z in $1, 2, \ldots, m - 1$. For our example, this issue is trivial. For realistically large values of a and m, however, implementation in a high-level language is a non-trivial issue because of the potential overflow associated with the product az. In particular, if $a = 16807$ and $m = 2^{31} - 1$ then 46 bits are required to hold the largest possible value of the az product.

In his original paper [26], Lehmer not only suggested the algebraic *form* of $f(\cdot)$, he also suggested that the (Mersenne) prime $m = 2^{31} - 1$ might be an appropriate choice for the modulus. For years this suggestion was largely ignored. Instead, programmers who knew more about code optimization than random number generation concentrated on the development of multiplicative generators with non-prime moduli of the form $m = 2^b$ where b was matched to the integer word size of the computer. The primary reason for this was execution speed. With a suitable choice of the multiplier a and some low-level programming the az product could be reduced to several shifts and adds and the mod m operation could be accomplished by *controlled* integer overflow [22, 31]. The result of this emphasis on speed was a generation of computationally efficient but highly non-portable and statistically flawed multiplicative linear congruential generators, the most notorious being the now infamous IBM SYSTEM/360 product RANDU [45].

Fortunately, by the mid-1960s a more balanced approach to random number generation had developed. Execution speed, while still important, was no longer the paramount issue [28] and the fundamental importance of a *prime* modulus was better appreciated—at least by specialists [16]. This, coupled with the development of 32-bit arithmetic as a computing standard, caused $2^{31} - 1 = 2147483647$ to reappear as an obvious, logical choice for the modulus. As the specialists began to standardize this choice, research shifted to a systematic search for good associated multipliers.

For a fixed prime modulus, in this case $m = 2^{31} - 1$, it is now clear that the systematic search for good associated multipliers involves finding a's which will pass each of the following three tests.

T_1: Is $f(z) = az \bmod m$ a full period generating function?

T_2: Is the full period sequence $\ldots, z_1, z_2, \ldots, z_{m-1}, z_1, \ldots$ random?

T_3: Can $f(\cdot)$ be implemented efficiently with 32-bit arithmetic?

The third section of this article is largely concerned with a theoretical discussion of these tests, with the primary emphasis on T_3 because it is the least well known of the three.

Each of the three T-tests effectively serves as a filter that limits the possible choices of a. Of the more than 2 billion possible choices corresponding to $m = 2^{31} - 1$, it is now known that only a relative handful of multipliers will pass all three tests. From this handful we have selected the multiplier 16807 to define a specific implementation of the minimal standard generator. Because of the subjective nature of T_2, however, there will always be some uncertainty about which multiplier is *best* and if this paper were to be written again in a few years it is quite possible that we would advocate a different multiplier based on the results of T_2 testing yet to be done.

The multiplier $a = 7^5 = 16807$ was first suggested by Lewis, Goodman and Miller in 1969 [27], based largely on the fact that

$$f(z) = 16807z \bmod 2147483647 \tag{5}$$

is a full period generating function. These authors also supplied evidence of randomness based on the results of what are now recognized to be relatively weak *empirical* statistical (T_2) tests. And, with regard to implementation, they presented an efficient but highly non-portable 32-bit SYSTEM/360 assembler language procedure.

In the intervening years more powerful *theoretical* tests of (full period) randomness have been developed, thereby strengthening T_2 significantly as discussed in [4, 9, 14, 18]. Extensive testing has revealed that the minimal standard generator defined by equation (5) also passes these tests—although not always with the best possible scores. Moreover, in 1979 Schrage [41] resolved the issue of machine independent implementation in a high-level language by demonstrating that equation (5) can be implemented correctly without overflow " . . . as long as the machine can represent all integers in the interval -2^{31} to $2^{31} - 1$."

In summary, $a = 16807$ and $m = 2^{31} - 1$ define a generator which has a full period, is demonstrably random, and can be implemented correctly on almost any system. The generator has been exhaustively tested and its characteristics are well understood, at least by specialists who frequently advocate its use [1, 22, 23, 39].

(Moreover, it has become a standard in the sense that it is now available in some commercial software packages such as subroutine RNUN in the IMSL library [17] and as subroutine DRAND in the simulation language SLAM II [42].) With all this in mind, we feel confident in recommending this random number generator as a minimal standard against which all others should be judged.

IMPLEMENTING THE MINIMAL STANDARD

The most obvious way to implement the minimal standard generator in a high-level language such as Pascal is as follows:

```
function Random : real;
                  (* Integer Version 1 *)
const
  a = 16807;
  m = 2147483647;
begin
  seed := (a * seed) mod m;
  Random := seed / m
end;
```

where

```
var seed : integer;
```

is a global variable used to hold the current value in the integer sequence of z's. This generator must be initialized by assigning `seed` a value between 1 and 2147483646. Random numbers uniformly distributed between 0 and 1 can then be generated as required via repeated calls to `Random`. Unfortunately, for most systems this version of `Random` is fatally flawed. Because the product `a * seed` can be as large as $16807 \times 2147483646 \approx 1.03 \times 2^{45}$, this version of `Random` is *not* a correct implementation of equation (5) unless `maxint` is $2^{46} - 1$ or larger. If `maxint` is smaller, as it will be for most contemporary systems, integer overflow will occur producing an error.

Any implementation of the minimal standard should be tested for correctness, *not* randomness. If the implementation is correct, tests for randomness are redundant; if the implementation is incorrect, it should be discarded. A simple but effective way of testing for a correct implementation is based on the fact that if $z_1 = 1$ then $z_{10001} = 1043618065$. In other words, `Random` is correct only if the program fragment

```
seed := 1;
for n := 1 to 10000 do
  u := Random;
Writeln('The current value
                  of seed is : ', seed);
```

produces 1043618065. If overflow is going to occur—thereby rendering incorrect all values from that point on, or causing program execution to halt—it will surely do so at some point in this sequence because intermediate values of `seed` as large as 2147483531 are produced.

Another obvious way to implement `Random` in Pascal is to do the integer calculations in real arithmetic, that is, declare the global variable

```
var seed : real;
```

and then use

```
function Random : real;
                    (* Real Version 1 *)
const
  a = 16807.0;
  m = 2147483647.0;
var
  temp : real;
begin
  temp := a * seed;
  seed := temp - m * Trunc(temp / m);
  Random := seed / m
end;
```

This version of `Random` will be correct if reals are represented with a 46-bit or larger mantissa (excluding the sign bit). For example, this version will be correct on all systems that support the IEEE 64-bit real arithmetic standard since the mantissa in that case is 53-bits. Note that there is a residual integer arithmetic requirement in this version of `Random`. The largest possible value of `Trunc(temp / m)`, however, is 16806 and no "integer out of range" error will occur provided `maxint` is at least this large.

Less obvious but extremely portable Pascal implementations of the minimal standard generator are possible based on Schrage's method first published in 1979 [41] and later refined in 1983 [1]. We present the integer implementation first and then the real version. The theory supporting these two implementations is presented in the next section.

The following integer version of `Random` is correct on any system for which `maxint` is $2^{31} - 1$ or larger. First declare `var seed : integer` and then use

```
function Random : real;
                  (* Integer Version 2 *)
const
  a = 16807;
  m = 2147483647;
  q = 127773;   (* m div a *)
  r = 2836;     (* m mod a *)
var
  lo, hi, test : integer;
begin
  hi := seed div q;
  lo := seed mod q;
  test := a * lo - r * hi;
  if test > 0 then
    seed := test
  else
    seed := test + m;
  Random := seed / m
end;
```

The essential feature of this implementation is that the variable `test` will never take on a value which cannot be represented correctly with 32 bits (including sign).

A functionally equivalent real version of this implementation is also possible. In general, this is the version that must be used if maxint is smaller than $2^{31} - 1$ (on some microprocessor systems). It will be correct on any system for which reals are represented with a 32-bit or larger mantissa (including the sign bit). First declare var seed : real and then use

```
function Random : real;
                    (* Real Version 2 *)
const
  a = 16807.0;
  m = 2147483647.0;
  q = 127773.0;   (* m div a *)
  r = 2836.0;     (* m mod a *)
var
  lo, hi, test : real;
begin
  hi := Trunc(seed / q);
  lo := seed - q * hi;
  test := a * lo - r * hi;
  if test > 0.0 then
    seed := test
  else
    seed := test + m;
  Random := seed / m
end;
```

There is also an integer arithmetic requirement in this implementation of the minimal standard. As with real version 1, however, the largest possible value of hi is 16807 and no errors will occur provided maxint is at least this large.

Over the years we (and our students) have implemented the minimal standard on a wide variety of systems. We have yet to encounter a contemporary system on which at least one of the four versions of Random could not be installed correctly. One system that deserves special mention is Turbo Pascal (version 3.0) from Borland International [46]. This popular system represents a good implementation exercise for two reasons: Turbo Pascal provides its own random number generator, which is not very good and maxint on this system is only $2^{15} - 1$, thereby eliminating the possibility of an efficient integer implementation. Fortunately plain Turbo Pascal uses 48-bit real arithmetic with a 40-bit mantissa and so it supports real version 2 of Random. In addition, the '87' version of Turbo Pascal (which uses the 8087 family of math coprocessors) uses IEEE standard 64-bit real arithmetic and so both real versions of Random work correctly with it. Real version 1 is more efficient than real version 2, by approximately a 2:3 ratio, but it is less portable.

One nagging implementation issue that must be resolved is how to handle seed. The method we have used herein—declaring seed to be global and updating its value by a side effect—is an obvious violation of good software engineering practice. Recognizing this, many authors advocate using seed as a formal var function parameter. We find this clumsy and misleading because statements like u := Random(seed) seem to suggest that the *user* should assign a value to seed and thereby create the random number. We believe that the user should always be able to *initialize* the generator by supplying an initial seed. Nevertheless, after initialization the ideal solution is to hide seed from the user. Unfortunately, standard Pascal does not support this approach.

There is also the question of what value should be used to initialize seed. Part of the folklore of random number generation is that some initial seeds are better than others—how many of us remember being told something like ... always use at least five digits with the last one odd [38]? For some generators this type of witchcraft is necessary. For the minimal standard, however, *all* initial seeds between 1 and 2147483646 are equally valid. If you use the minimal standard generator to produce simulation results and if these results prove to be very sensitive to the choice of initial seed then distrust your simulation, not the generator.

We recommend the construction of an initialization procedure, Randomize, which prompts for an initial value of seed and forces it to be an integer between 1 and $2^{31} - 2$. As a convention we frequently advise our students to respond with the 9 digits of their social security number. This helps insure the statistical independence of each student's results and provides a sometimes desirable standardization. One advantage to using the minimal standard is that results can be reproduced if desired. This is possible, however, only if the initial seed is remembered. For the same reason we view any implementation of Randomize as unacceptable if it does not allow the user to supply the initial seed. Default initial values, particularly those supplied by an inaccessible program counter or system clock, should be used *only* when the user chooses not to supply one.

THEORY

The purpose of this section is to present some theoretical details in support of the previous discussion. We begin with a discussion of T_1—the test for full period. We assume, as before, that $f(z) = az \bmod m$ where m is a large prime, that $2 \le a \le m - 1$ and that the sequence $z_1, z_2, z_3, \ldots$ is generated iteratively as $z_{n+1} = f(z_n)$ with the initial seed z_1 in $1, 2, \ldots, m - 1$. We also make use of the following basic results [18, p. 10 and p. 375]

$$z_{n+1} = f(z_n) = a^n z_1 \bmod m \quad n = 1, 2, 3, \ldots \tag{6}$$

$$a^{m-1} \bmod m = 1 \tag{7}$$

and a standard number theoretic definition: If m is prime then a is a *primitive element modulo m* (or *primitive root of m*) iff $a^n \bmod m \neq 1$ for $n = 1, 2, \ldots, m - 2$ [18, p. 19]. Equation (7) is a classic result known as Fermat's Theorem.

From equation (6), if $z_1 = 1$ then in general $z_{n+1} = a^n \bmod m$. Let p be the smallest value of n such that $z_{p+1} = 1$. The existence of $p \le m - 1$ is guaranteed by

Fermat's Theorem. Because $z_{p+1} = z_1$, it follows that $z_{p+2} = z_2$ and in general $z_{p+n} = z_n$ for all $n \geq 1$; that is, the sequence of z's is *periodic* with period p. If $p = m - 1$ then $f(\cdot)$ is a *full period* generator. Clearly this is true iff a is a primitive root of m. (On the other hand, if a is not a primitive root of m then $f(\cdot)$ is not a full period generator and $p < m - 1$ must be a divisor of $m - 1$.) Relative to the example Lehmer generator, we see that both 6 and 7 are primitive roots of 13 and that 5 is not. With regard to the primitive roots of $m = 2^{31} - 1$, the following is known [9, 27]:

(1) there are a total of 534,600,000 primitive roots;[1]

(2) the smallest primitive root is 7;

(3) a is a primitive root iff $a = 7^b \bmod m$ where b is relatively prime to $m - 1$;

(4) the prime factors of $m - 1$ are 2, 3, 7, 11, 31, 151 and 331;

(5) $b = 5$ is relatively prime to $m - 1$ and thus $7^5 = 16807$ is a primitive root.

These results completely resolve the use of T_1 testing, at least in theory: $f(\cdot)$ is a full period generator with $m = 2^{31} - 1$ iff a is one of the approximately 534 million associated primitive roots. Thus T_1 is a filter that eliminates approximately 75 percent of the possible multipliers. Several years ago, Fishman and Moore [9] took on the Herculean task of T_2-testing the remaining 25 percent for randomness. The following briefly summarizes their results.

It is well-known that all linear congruential generators suffer from the inherent flaw that, in 3-space for example, the points (z_1, z_2, z_3), (z_2, z_3, z_4), (z_3, z_4, z_5) all fall on a finite—and possibly small—number of parallel (hyper)planes [30]. All of the most powerful theoretical T_2 tests are based on analyzing the *uniformity* of this hyperplane (lattice) structure in k-space for small values of k. As the authors stated, "[we] regard a multiplier as optimal if for $2 \leq k \leq 6$ and each set of parallel hyperplanes the Euclidean distance between adjacent hyperplanes does not exceed the minimal achievable distance by more than 25 percent. . . . [and] among the more than 534 million full period multipliers examined in this study, our research identified only 414 optimal multipliers." [9].

We should mention that due to a bookkeeping error in [9] the number 414 is incorrect, there are actually only 410 optimal multipliers. Nonetheless, this error in no way diminishes the significance of the work. Also, only 205 of the 410 multipliers were actually published. The other 205 can be calculated since primitive roots occur in pairs, that is, for $m = 2^{31} - 1$ the multiplier $a = 7^b \bmod m$ is a primitive root iff $a' = 7^{m-1-b} \bmod m$ is also. Finally, and this is most important, there is nothing magic about 410. If the 25 percent distance criteria is increased slightly, significantly more than 410 optimal multipliers will result [9].

[1] 534,600,000 is exact; the corresponding value in [9] is approximate.

The multiplier 16807 is not in Fishman and Moore's list of 410 optimal multipliers. For that matter, the other multipliers (630,360,016 [37] and 397,204,094 [14]) commonly used with $2^{31} - 1$ are not in the list. This certainly does not mean that 16807 is an unsatisfactory multiplier; it just means that the sequence 16807 produces is not quite as random as some others. It is important to appreciate that the results in [9] were derived *independent* of implementation (T_3) considerations and in that sense the results are incomplete. We now turn to a discussion of 32-bit implementation using Schrage's method [1]. Following that, we will have more to say about the results in [9].

The basic idea is to construct an algorithm that will evaluate $f(z) = az \bmod m$ in such a way that all intermediate results will be bounded by $m - 1$. A 32-bit implementation then follows immediately if $m \leq 2^{31}$. We begin by observing that the potential overflow associated with $f(\cdot)$ is caused because the product az is formed prior to the mod by m and that the overflow could be avoided if the order of these two operations were reversed. This could be accomplished trivially if it were possible to factor m as $m = aq$ for some integer q. If so then $f(\cdot)$ could be written as $f(z) = az \bmod aq = a(z \bmod q)$. Of course m is prime and no such factorization is possible. It is possible, however, to approximately factor m as follows

$$m = aq + r \tag{8a}$$

where

$$q = m \operatorname{div} a \quad \text{and} \quad r = m \bmod a. \tag{8b}$$

If the remainder r is small, specifically if $r < q$, this decomposition of m enables us to construct an algorithm for evaluating $f(z)$ without producing intermediate results larger than $m - 1$ in magnitude. Note that in general the quotient satisfies $q \geq 1$ and that $1 \leq r \leq a - 1$. In the particular case of $a = 16807$ and $m = 2^{31} - 1$, we have $q = 127{,}773$ and $r = 2836$.

A mathematically equivalent expression for $f(z) = az \bmod m = az - m(az \operatorname{div} m)$ can be derived by first adding and subtracting $m(z \operatorname{div} q)$ and then doing some algebra to yield

$$f(z) = \gamma(z) + m\delta(z) \tag{9a}$$

where

$$\gamma(z) = a(z \bmod q) - r(z \operatorname{div} q) \tag{9b}$$

and

$$\delta(z) = (z \operatorname{div} q) - (az \operatorname{div} m). \tag{9c}$$

It can be shown that if $r < q$ then for all z in $1, 2, \ldots, m - 1$ the following are true:

(1) $\delta(z)$ is either 0 or 1

(2) both $a(z \bmod q)$ and $r(z \operatorname{div} q)$ are in $0, 1, 2, \ldots, m - 1$

(3) $|\gamma(z)| \leq m - 1$.

Item (3) is an obvious consequence of (2) and (2) follows from equation (8) and the assumption that $r < q$. Item

(1) is a consequence of the fact that if x and y are real numbers with $0 \le x - y \le 1$ then $\lfloor x \rfloor - \lfloor y \rfloor$ is either 0 or 1 where $\lfloor \cdot \rfloor$ denotes the usual greatest integer function.

The key to Schrage's method is that the operation which would cause an overflow is trapped in $\delta(z)$ and item (1) provides a mechanism for inferring the value of $\delta(z)$ from a knowledge of $\gamma(z)$. Specifically, because $1 \le f(z) \le m - 1$ it follows from equation (9a) that $\delta(z) = 0$ iff $1 \le \gamma(z) \le m - 1$ and that $\delta(z) = 1$ iff $-(m - 1) \le \gamma(z) \le -1$. Thus, the evaluation of $\delta(z)$ is not necessary. Instead, to evaluate $f(z)$, first evaluate $\gamma(z)$. Then if $\gamma(z) > 0$ assign $f(z) := \gamma(z)$ else assign $f(z) := \gamma(z) + m$. This is the algorithm implemented in version 2 of Random.

Note that a slightly modified version of Random could be used instead based on the following algorithm. Evaluate $\alpha(z) = a(z \bmod q)$ and $\beta(z) = r(z \operatorname{div} q)$. Then if $\alpha(z) > \beta(z)$ assign $f(z) := \alpha(z) - \beta(z)$ else assign $f(z) := \alpha(z) + (m - \beta(z))$. Because the logic of this algorithm is more obscure, we have chosen not to use it. This algorithm does have the aesthetic property that (consistent with $(\cdot) \bmod m$) $\alpha(\cdot)$ and $\beta(\cdot)$ are bound to $0, 1, 2, \ldots, m - 1$.

In the particular case $m = 2^{31} - 1$ we take the condition $(m \bmod a) < (m \operatorname{div} a)$ as the *definition* of test T_3. Based on this test, all the other multipliers commonly used with $m = 2^{31} - 1$ can be rejected immediately. The question then is how many multipliers are like 16807, that is, how many multipliers pass both T_1 and T_3? We have verified by exhaustive search that the answer is 23,093 and that *none* of these multipliers are in Fishman and Moore's list of 410. Thus, in terms of finding multipliers which pass *all three* T-tests, Fishman and Moore's 25 percent distance criteria is excessively restrictive.

Two obvious questions remain: Which of the 23093 multipliers have the best T_2 scores and do any of these full period, 32-bit compatible multipliers have a T_2-score significantly higher than 16807? The recent article by L'Ecuyer [24] provides a partial answer to these questions. This article reports the results of a search to find the best full period multiplier a such that $a^2 < m$. Although the test $a^2 < m$ is a restrictive version of our more general $r < q$ test (only 11465 of the 23093 multipliers satisfy $a^2 < 2^{31} - 1$), L'Ecuyer was still able to find several full period multipliers with better T_2-scores than 16807. Of these, he reported $a = 39{,}373$ as best.

In a very recent more comprehensive search Fishman subjected *all* of our 23093 multipliers to the same T_2-tests as in [9]—with a 30 percent distance criteria. He found several good 32-bit compatible multipliers. Of these, $a = 48{,}271$ and $a = 69{,}621$ appear to be best. Both of these are in the set of (11628 = 23093 − 11465) 32-bit compatible multipliers not tested by L'Ecuyer and both have better T_2-scores than 39373 and 16807.

So then, which 32-bit compatible multiplier should be used? Our guess is that at some future point we will switch to either $a = 48271$ (with $q = 44{,}488$ and $r = 3399$) or $a = 69621$ (with $q = 30{,}845$ and $r = 23{,}902$). We are still awaiting the results of further testing and the accumulation of favorable user experience. For now, we feel comfortable continuing to use $a = 16807$.

A SAMPLING OF INADEQUATE GENERATORS

In this section we present selected examples of inadequate random number generators that have either appeared in recently published computer science textbooks or are currently supplied by popular programming environments. To simplify the discussion we restrict the examples to multiplicative and mixed linear congruential generators. This is the class of generators for which the theory is most complete.

Many multiplicative linear congruential generators are descendents of the infamous RANDU [45] defined by

$$f(z) = 65539z \bmod 2^{31}. \qquad (10)$$

This generator was first introduced in the early 1960s; its use soon became widespread. In retrospect RANDU was a mistake. The non-prime modulus was selected to facilitate the mod operation and the multiplier, which is $2^{16} + 3$, was selected primarily because of the simplicity of its binary representation. Research and experience has now made it clear that RANDU represents a flawed generator with no significant redeeming features. It does not have a full period and it has some distinctly non-random characteristics. Knuth calls it "really horrible" [18, p. 173].

In general, any multiplicative linear congruential generator with modulus $m = 2^b$ is flawed in the sense that it can not have a full period; instead the maximum possible period is only $2^{b-2} = m/4$. This maximum period is achieved iff $(a \bmod 8)$ is either 3 or 5 and the initial seed is an odd integer. If these conditions are met the generator will produce a periodic permutation of half the odd integers between 1 and $2^b - 1$ [7].

For historical reasons, implementations of multiplicative generators with $m = 2^b$ frequently make use of wordlength and language dependencies. The following 32-bit Fortran RANDU fragment is typical:

```
SEED = 65539 * SEED
IF (SEED .LT. 0) SEED
   = (SEED + 2147483647) + 1
```

It is difficult to find two lines of code which violate more software engineering principles—the intent is obscure and the result is non-portable. The first line makes use of controlled integer overflow to MOD the product by 2^{32} and the second line clears the sign bit if necessary by adding 2^{31}. The net result is a MOD by 2^{31}. This two line implementation of equation (10) is correct *only* on 32-bit, two's complement systems for which integer overflow is not a fatal error. Programming like this might have been considered clever and efficient 25 years ago, but not today.

Unfortunately, the 1982 simulation text by Payne [36] and the 1985 scientific programming text by Nyhoff and Leestma [35] both present a controlled overflow implementation of RANDU as the generator of choice. Worse, Gottfried's 1985 Pascal text [12] recom-

mends the 16-bit microprocessor RANDU analogue defined by $a = 2^8 + 3$ and $m = 2^{15}$, again with controlled overflow and instructions about how to disable overflow checking. Because of its widespread use at the time, RANDU was commonly found in the literature of the 1960s and early 1970s. The inadequacies of this generator are now so well known, however, that it should never be recommended in the computer science literature of the 1980s.

It is well known that RANDU's T_2 scores can be improved somewhat by changing the multiplier [18, p. 104]. With this in mind, some people have naively attempted to improve things by changing the multiplier to 16807. For example, function Random in Prime Sheffield Pascal—another system commonly used by our students—is a multiplicative generator with $a = 16807$ and $m = 2^{31}$ [11]. Unfortunately, (16807 mod 8) is 7 and so

$$f(z) = 16807z \bmod 2^{31} \tag{11}$$

does not even have the maximum possible period! Moreover, this generating function is apparently so non-random that at least one software vendor has felt it is necessary to shuffle the output to produce acceptable randomness. Specifically, subroutine UNIFORM in the statistical package SAS is based on equation (11) with an added shuffle to randomize the output [38]. We should add that the use of a generator based on equation (11) in Prime Sheffield Pascal is particularly unfortunate because the PRIMOS operating system [44] provides a much better generator, RAND$A, which is the minimal standard.

When it comes to constructing random number generators, there is no such thing as getting the software *almost* right. As an illustration, the 1985 Fortran text by Smith [43] presents a flawed generator presumably based on a failed attempt to implement a 16807 multiplicative generator using controlled overflow. The two relevant lines of code are:

```
SEED = 16807 * SEED
IF (SEED .LT. 0) SEED
   = SEED + 2147483647
```

If the author had added a 1 in the second line, the result would have been a 32-bit two's complement implementation of equation (11). Or perhaps the author erroneously assumed that omitting the 1 would yield a correct controlled overflow implementation of equation (5). In any case, as presented, this generator is not a modified version of RANDU and it is not the minimal standard. We tested this generator on a 32-bit two's complement system using 150 different initial seeds. In each case the resulting sequence of z's ultimately collapsed into one of three disjoint periodic subsequences of length 21,729 or 11,330 or 9,181 depending on the initial seed.

We view any generator as inadequate by inspection if its period is too small. For example the 1980 simulation text by Maryanski [32] recommends a multiplicative generator defined by $a = 20{,}403$ and $m = 2^{15}$. The period of this generator is $2^{13} = 8192$, which is far too small for any serious simulation activity. Similarly, a 1985 Modula-2 system reference manual [29] presents a Lehmer generator with $a = 13$ and $m = 2311$. The modulus is tiny, the multiplier is not a primitive root of the modulus and the resulting period is just 1155. An even worse example can be found in the 1985 LISP text by Gabriel [10] which uses $a = 17$ and $m = 251$. Again, the multiplier is not a primitive root of the modulus and the resulting period in this case is just 125.

As a final multiplicative generator example, the 1982 simulation text by Bulgren [2] presents an out-of-date generator with $a = 5^{13}$ and $m = 2^{35}$. Although the period is 2^{33} and the T_2-scores are marginally acceptable (see [18, p. 103]), the associated Fortran implementation relies on a 36-bit controlled overflow technique which is unlikely to be correct on any contemporary system.

Mixed linear congruential generators [18, p. 10] are generalizations of multiplicative generators with generating functions of the form $f(z) = (az + c) \bmod m$ where the additive parameter c satisfies $c \bmod m \neq 0$. At the expense of one extra addition per random number, the effect of c is to allow $z = 0$ as a possible value and thus a full period sequence has length m rather than $m - 1$. In theory, the modulus m can be any positive integer, prime or non-prime. In practice, however, m is usually either a power of 2 or a power of 10.

In the 1960s there was some hope that mixed generators would prove to be *better* than multiplicative generators (like RANDU), particularly if $m = 2^b$. In retrospect, this has not proven to be true and today, with one notable exception [18, p. 170], mixed generators are rarely ever recommended by specialists. They are, however, favored by many textbook authors and some system programmers, presumably because the T_1-test for full period is so easy to apply. Specifically, if $m = 2^b$ then a mixed generator has full period iff $a \bmod 4 = 1$ and c is odd [18, p. 20]. For other values of m the test is only slightly more difficult to apply [18, p. 16]. In a sense it is unfortunate that this test for full period is so trivial as it falsely encourages non-specialists to build their own generators. The generator discussed next is a good illustration of this.

In 1978 P. Grogono published an introductory Pascal text and included in it a random number generator defined by

$$f(z) = (25173z + 13849) \bmod 2^{16}. \tag{12}$$

This popular text is now in its second edition [13] and the generator remains essentially unchanged. To its credit, this mixed generator has a full period (of length 65536) and, because the largest possible value of $25173z + 13849$ is $\approx 1.54 \times 2^{30}$ it can be easily implemented correctly provided maxint is $2^{31} - 1$ or larger. Since it contains a relatively small period, this generator is unsatisfactory for scientific applications. Moreover, we are not aware of any published T_2-test results which suggest that this generator's output is adequately random. Despite this, Grogono's generator has spread to

many other recent textbooks [6, 25, 33, 34] and is now something of an emerging standard in the undergraduate computer science textbook market.

Most of the other mixed generators found in computer science texts are even less satisfactory than Grogono's generator. We only list a few of these; the reader is encouraged to look for the others—they are relatively easy to find. The 1986 text by Lamb [20] and the 1987 text by Konvalina and Wileman [19] present mixed generators with $a = 10924$, $c = 11830$, $m = 2^{15} + 1$ and $a = 93$, $c = 1$, $m = 2^{13}$ respectively. In each case the period is full but too small. The same comment applies to the 1987 text by Clocksin and Mellish [3] which suggests $a = 125$, $c = 1$ and $m = 2^{12}$. The 1984 text by Savitch [40] and the 1987 text by Lamie [21] present full period mixed generators with even smaller periods. These generators are defined by $a = 40$, $c = 3641$, $m = 729$ and $a = 61$, $c = 2323$, $m = 500$ respectively and both generators contain an obfuscation; in each case c can, and should, be replaced with c mod m.

The 1986 text by Collins [5] presents a prime modulus mixed generator with an insidious flaw that requires special comment. The generator is defined by

$$f(z) = (9806z + 1) \bmod 131071 \qquad (13)$$

where 131,071 is the Mersenne prime $2^{17} - 1$. It is easily verified, however, that $f(37911) = 37911$ which means that if by chance 37911 is used as an initial seed, the generator will be stuck on this value forever! Also, it can be verified that if *any* other initial seed between 0 and 131070 is used the generator will appear to work correctly and cycle with an (almost full) period of 131070.

The binary representation of a sequence of z's produced by Grogono's generator reveals an interesting pattern. The least significant bit cycles with period 2, the next most significant bit cycles with period 4, the next cycles with period 8 and so forth, and only the most significant bit cycles with a full period. It turns out that this distinctly non-random behavior is a characteristic of *all* full period mixed generators with $m = 2^b$ [18, p. 12]. For example, the UNIX® operating system supports a full period mixed generator called `rand`. This generator is defined by

$$f(z) = (1103515245z + 12345) \bmod 2^{31} \qquad (14)$$

and its deficiencies are well known—according to Berkeley 4.2 documentation, the low bits of the numbers generated are not very random. Similarly, the random number generator in standard Turbo Pascal is a full period mixed generator defined by

$$f(z) = (129z + 907633385) \bmod 2^{32} \qquad (15)$$

with the output normalized via division by 2^{32}. This generator also exhibits small period cycling in its least significant bits. Worse, the multiplier is too small and was apparently chosen for the wrong reason—the simplicity of its binary representation. (Incidentally, the generator in '87' Turbo is subtly different. The output is normalized via division by 2^{31} rather than 2^{32} and if the result, u, is greater than 1, then the output is $2 - u$ instead.)

There is really no argument in support of *any* of the example generators cited in this section. Most of them are naive implementations of a deceptively simple idea. All should be discarded. Even the best of them, those which have a full period and are correctly implemented, are inferior to the minimal standard.

CONCLUDING REMARKS

Many computer scientists will never have more than an occasional need to use a random number generator. And on those occasions the statistical *goodness* of the random numbers they generate may not be of paramount importance. For some of us, however, convenient and frequent access to a verifiably good random number generator is fundamentally important. If you have need for a random number generator, particularly one that will port to a wide variety of systems, and if you are not a specialist in random number generation and do not want to become one, use the minimal standard. It should not be presumed that it is easy to write a better one.

For those interested in learning more about random number generation we recommend, in addition to Knuth [18], the simulation texts by Fishman [7], Law and Kelton [22], and Bratley, Fox and Schrage [1]. The discussion in [1] is particularly relevant to this article. We also recommend the articles by L'Ecuyer [24] and Wichmann and Hill [47]. The *combined* generators proposed in these articles represent a logical extension of the minimal standard. These generators appear to yield better statistical properties in those (rare) situations when the minimal standard alone may be inadequate.

Acknowledgments. Our thanks to George Fishman who kindly agreed to T_2-test our list of 32-bit compatible multipliers, to John Burton who helped compile this list, and to Paul Stockmeyer, Phil Kearns, and Don Lansing who read and commented on a preliminary draft of the paper.

UNIX is a registered trademark of Bell Laboratories.

REFERENCES

1. Bratley, P., Fox, B.L., and Schrage, E.L. *A Guide to Simulation.* Springer-Verlag, New York, 1983, pp. 180–213.
2. Bulgren, W.G. *Discrete System Simulation.* Prentice-Hall, Englewood Cliffs, N.J., 1982, p. 155.
3. Clocksin, W.F., and Mellish, C.S. *Programming in Prolog.* Springer-Verlag, New York, 1987, p. 153.
4. Coveyou, R.R., and MacPherson, R.D. Fourier analysis of uniform random number generators. *J. ACM 14* (Jan. 1967), 100–119.
5. Collins, W.J. *Intermediate Pascal Programming: A Case Study Approach.* McGraw-Hill, New York, 1986, p. 157.
6. Cooper, D., and Clancy, M. *Oh! Pascal!* 2nd Ed. W. W. Norton, New York, 1985, p. 145.
7. Fishman, G.S. *Principles of Discrete Event Simulation.* Wiley-Interscience, New York, 1978, pp. 345–391.
8. Fishman, G.S., and Moore, L.R. A statistical evaluation of multiplicative congruential random number generators with modulus $2^{31} - 1$. *J. Am. Stat. Assoc. 77*, 377 (Mar. 1982), 129–136.
9. Fishman, G.S., and Moore, L.R. An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$. *SIAM J. Sci. Stat. Comput. 7*, 1 (1986), 24–45.

10. Gabriel, R. *Performance and Evaluation of LISP Systems.* M.I.T. Press, Cambridge, Mass., 1985, p. 140.
11. Gilbert, J.R. *The University of Sheffield Pascal System for Prime Computers.* University of Sheffield, Sheffield, England, 1987, p. 10.
12. Gottfried, B.S. *Schaum's Outline of Theory and Problems of Programming with Pascal.* McGraw-Hill, New York, 1985, p. 143.
13. Grogono, P. *Programming in Pascal.* 2nd Ed. Addison-Wesley, Reading, Mass., 1984, pp. 135–137.
14. Hoaglin, D.C. Theoretical properties of congruential random-number generators: An empirical view. Memo NS-340. Dept. of Statistics, Harvard University, Cambridge, Mass., 1976.
15. Hull, T.T., and Dobell, A.R. Random number generators. *SIAM Rev. 4* (July 1962), 230–254.
16. Hutchinson, D.W. A new uniform pseudorandom number generator. *Commun. ACM 9*, 6 (June 1966), 432–433.
17. *IMSL Stat/Library User's Manual.* IMSL, Houston, Tex., 1987, pp. 947–951.
18. Knuth, D.E. *The Art of Computer Programming.* 2nd Ed. Addison-Wesley, Reading, Mass., 1981.
19. Konvalina, J., and Wileman, S. *Programming with Pascal.* McGraw-Hill, New York, 1987, p. 288.
20. Lamb, R. *Pascal Structure and Style.* Benjamin/Cummings, Menlo Park, Calif., 1986, pp. 226–227.
21. Lamie, E.L. *Pascal Programming.* John Wiley and Sons, New York, 1987, p. 150.
22. Law, A.M., and Kelton, W.D. *Simulation Modeling and Analysis.* McGraw-Hill, New York, 1982, pp. 219–239.
23. Lavenberg, S.S., Ed. *Computer Performance Modeling Handbook.* Academic Press, New York, 1983, pp. 223–229.
24. L'Ecuyer, P. Efficient and portable combined random number generators. *Commun. ACM 31*, 6 (June 1988), 742–749, 774.
25. Leestma, S., and Nyhoff, L. *Pascal Programming and Problem Solving.* Macmillan, New York, 1984, pp. 172–173.
26. Lehmer, D.H. Mathematical methods in large-scale computing units. *Annu. Comput. Lab. Harvard Univ. 26* (1951), 141–146.
27. Lewis, P.A., Goodman, A.S., and Miller, J.M. A pseudo-random number generator for the System/360. *IBM Syst. J. 8*, 2 (1969), 136–146.
28. Maclaren, M.D., and Marsaglia, G. Uniform random number generators. *J. ACM 12*, 1 (Jan. 1965), 83–89.
29. *MacModula-2 System Reference Manual.* Modula Corporation, 1985, p. 41.
30. Marsaglia, G. Random numbers fall mainly in the planes. *Natl. Acad. Sci. Proc. 61* (Sept. 1968), 25–28.
31. Marsaglia, G., and Bray, T.A. One-line random number generators and their use in combinations. *Commun. ACM 11*, 11 (Nov. 1968), 757–759.
32. Maryanski, F. *Digital Computer Simulation.* Hayden, Rochelle Park, N.J., 1980, pp. 224–230.
33. Moffat, D.V. *Common Algorithms in Pascal.* Prentice-Hall, Englewood Cliffs, N.J., 1984, pp. 201–203.
34. Molluzzo, J.C., and Buckely, F. *Discrete Mathematics.* Wadsworth, Belmont, Calif., 1986, pp. 219–221.
35. Nyhoff, L., and Leestma, S. *FORTRAN 77 for Engineers and Scientists.* Macmillan, New York, 1985, pp. 292–294.
36. Payne, J.A. *Introduction to Simulation: Programming Techniques and Methods of Analysis.* McGraw-Hill, New York, 1982, pp. 105–106.
37. Payne, W.H., Rabung, J.R., and Bogyo, T.P. Coding the Lehmer pseudo-random number generator. *Commun. ACM 12*, 2 (Feb. 1969), 85–86.
38. *SAS User's Guide: Basics, Version 5 Edition.* SAS Institute Inc., Cary, N.C., 1985, pp. 278–280.
39. Sauer, C.H., and Chandy, K.M. *Computer Systems Performance Modeling.* Prentice-Hall, Englewood Cliffs, N.J., 1981, pp. 195–199.
40. Savitch, W.J. *Pascal, An Introduction to the Art and Science of Programming.* Benjamin/Cummings, Menlo Park, Calif., 1984, p. 244.
41. Schrage, L. A more portable FORTRAN random number generator. *ACM Trans. Math. Softw. 5*, 2 (June 1979), 132–138.
42. *SLAM II Installation and Operations Guide, Version 3.2.* Pritsker and Associates, West Lafayette, Ind., 1986, pp. 3.4–3.9.
43. Smith, M. *FORTRAN 77, A Problem-Solving Approach.* Houghton Mifflin, Boston, Mass., 1985, pp. 330–331.
44. *Subroutines Reference Guide.* 3rd Ed. Prime Computer, Natick, Mass., 1984, p. 12.45.
45. *System/360 Scientific Subroutine Package, Version III, Programmer's Manual.* IBM, White Plains, New York, 1968, p. 77.
46. *Turbo Pascal, Version 3.0.* Borland International, Scotts Valley, Calif., 1986.
47. Wichmann, B.A., and Hill, I.D. An efficient and portable pseudo-random number generator. *Appl. Stat. 31* (1982), 188–190.

CR Categories and Subject Descriptors: G.3 [**Probability and Statistics**]: Random number generation; G.4 [**Mathematical Software**]: Portability
General Terms: Algorithms, Standardization, Theory
Additional Key Words and Phrases: Lehmer generators, simulation

Received 9/87; accepted 2/88

ABOUT THE AUTHORS:

STEVE PARK is a professor of computer science at the College of William and Mary. Prior to this position, he was involved in aerospace research at NASA, Langley Research Center. His research interests include modeling and simulation, with an emphasis on the conceptual design and performance analysis of digital imaging systems. Author's present address: Stephen K. Park, Department of Computer Science, College of William and Mary, Williamsburg, VA 23185.

KEITH MILLER teaches computer science at the College of William and Mary in Virginia, and consults for Computer Sciences Corporation, NASA Langley Research Center, and the C&P Phone Company. His research interests include software engineering, abstract data types, image data structures, and computer ethics. Author's present address: Keith Miller, Department of Computer Science, College of William and Mary, Williamsburg, VA 23185.