TECHNICAL CORRESPONDENCE

REMARKS ON CHOOSING AND IMPLEMENTING RANDOM NUMBER GENERATORS

This correspondence is motivated by two articles in *Communications,* one of them proposing an "industry standard" random number generator [10], the other describing machine language implementations of that generator [2]. In the first article, under the title "Random number generators: good ones are hard to find," Park and Miller confine their discussion to congruential generators and advocate the generator $x_n = 16807x_{n-1} \bmod 2^{31} - 1$ as the "good one," citing some 16-bit generators or the notorious 32-bit RANDU for comparison as the "bad one."

Such is the impact of *Communications,* with its wide readership, that numerous people have the impression that $x_n = 16807 \bmod 2^{31} - 1$ is the only good random number generator—the rest are bad.

The principal points I want to make are:

- The congruential generator $x_n = 16807x_{n-1} \bmod 2^{31} - 1$ is a good generator, but not a great generator. There are many more promising ones.
- The redeeming feature of the generator is the closeness of its modulus to a power of 2, providing nice, but tricky, machine language implementations. But for such implementations, moduli $2^{32} - 2$ or $2^{32} - 5$ seem better choices.
- If double precision or other means for extended precision integer arithmetic is to be used, there are many choices better than $2^{31} - 1$ for the modulus of the generator.
- Most congruential generators work quite well for small simulations, but their periods are far too short for modern needs and the tendency is toward other kinds of generators with extremely long periods.
- If a generator is to be adopted as an "industry standard," there is no consensus that it should be a congruential generator and far from a consensus that it should be $x_n = 16807x_{n-1} \bmod 2^{31} - 1$.

Residue systems

Most modern CPU's treat integers as 16- or 32-bit words. Examples with strings of 16 or 32 bits are tedious and difficult to follow, so we will assume a simple CPU with integers of 4 bits. The ideas readily extend to 16-, 32- or even the anticipated 64-bit CPU's.

So, assume a two's-complement CPU with integer words of 4 bits, and thus there are 16 possible bit patterns. For some applications it is useful to use this set of residues to identify bit patterns, the least-positive residue (LPR) system:

0000	0001	0010	0011	0100	0101	0110	0111
0	1	2	3	4	5	6	7
1000	1001	1010	1011	1100	1101	1110	1111
8	9	10	11	12	13	14	15

Such a system might be used internally, for example, for designating memory locations. Many high-level language implementations use the analogue of the following set of representations, the least-absolute residue (LAR) system:

0000	0001	0010	0011	0100	0101	0110	0111
0	1	2	3	4	5	6	7
1000	1001	1010	1011	1100	1101	1110	1111
−8	−7	−6	−5	−4	−3	−2	−1

It is important to note that the CPU doesn't "know" what residue system is intended; it is wired to produce a double-word bit pattern for arithmetic operations on pairs of one-word integer bit patterns. Addition or multiplication of two integers produces a double word pattern with the rightmost word usually considered the result. Thus multiplying 0101 by 1001 produces the double work 00101011, with the rightmost word the "answer." This is interpreted in the LPR system as $5 \times 9 = 13$, which is ordinary arithmetic modulo 16, while in the LAR system this is interpreted as $5 \times -7 = -3$, again arithmetic modulo 16, but using least absolute residues. Addition of two integer words works in a similar way —a double word is formed with the rightmost word taken as the "answer." Thus adding 1100 to 1101 produces 0001 1001 with the rightmost word 1001 taken as the answer. In the LPR system this is interpreted as $12 + 13 = 9$, while the LAR interpretation is $(-4) + (-3) = -7$; in either case, the residue of 2^4 is appropriate for that particular residue system. Results are similar for modern 32-bit (or 16-bit) CPU's: integer arithmetic produces the bit patterns consistent with arithmetic modulo 2^{32} (or 2^{16}), with interpretation in a particular residue system left to the user.

Note that the default settings of some compilers call for "integer overflow" messages when the sum or product of apparently positive numbers produces a negative number, and program switches may have to be set to prevent interrupts. No such problems occur with machine language instructions, and the programmer may exploit the way the instructions are "wired," free to use whatever residue system he chooses —lease absolute, least positive or, as we shall see in the following, some other interpretation of the way bit patterns are produced and identified.

Applications

As was pointed out in *Communications* in 1968 [5], extremely fast implementations of congruential random number generators result from exploiting the modulo 2^{32} arithmetic inherent in two's complement CPU's. Thus the single Fortran instruction $I = 69069*I$ or Pascal $I := 69069*I$ will cause the current (signed) random integer I to be replaced by the next element of the sequence generated by $x_n = 69069*x_{n-1} \bmod 2^{32}$. This is the system generator for Vax's and part of the combination generator in the widely used McGill Super Duper generator for IBM mainframes. Points in higher dimensions with coordinates from the 69069 generator fall on a lattice, as must those of all congruential generators [4,6], but some lattices are better than others, and 69069 was chosen for its good lattices. It should be noted that the lattices of the proposed "industry standard" $x_n = 16807x_{n-1} \bmod 2^{31} - 1$ are not very good ones; those of most any "randomly" chosen multiplier such as 314159269 are much better for that modulus.

Modulus $2^{31} - 1$ vs. Modulus $2^{32} - 2$

The generator advocated by Park and Miller [10] as a "minimal standard" random number generator is the congruential generator $x_n = 16807x_{n-1} \bmod p$ for the prime modulus $p = 2^{31} - 1$, or, more generally, $x_n = ax_{n-1} \bmod p$ for any primitive root a of p. Arithmetic modulo p is not easily implemented. Park and Miller suggest using double precision or use of computers able to handle integers of up to 50 bits or fairly elaborate programs that perform extended-precision arithmetic by breaking integers into parts then doing arithmetic on, before combining, the parts. But, there are more desirable prime moduli for extended-precision implementations. The closeness of $2^{31} - 1$ to a power of 2 is its redeeming (and damning) feature, fully exploitable only through machine language implementations.

We now describe a method for exploiting that feature through integer arithmetic. A previous old, as well as a recent new article in the *Communications* have discussed such implementations: Payne, Rabung and Bogyo [11] for IBM 360s in 1969 and Carta in 1990 [2] for unspecified CPU's that seem to do integer arithmetic with 31-bit registers, a variety unknown to us. Our new proposal has several advantages: it leads to faster and easier implementations and it produces both positive and negative random integers for high-level language use.

We will use modulus $2^{32} - 2$ rather than $2^{31} - 1$. Thus our generator is $x_n = 16807x_{n-1} \bmod 2^{32} - 2$, or, more generally, $x_n = ax_{n-1} \bmod 2^{32} - 2$ for any *odd* primitive root a of the prime $2^{31} - 1$. Let x be the current random number, a 32-bit word. Form the product $a \times x$ in adjoining 32-bit words. Call it y. We may view y as $y = 2^{32}t + b$, where t is the integer in the top-half, and b the integer in the bottom-half of the two-word product. We want y modulo $2^{32} - 2$. Writing $y = 2^{32}t + b = t(2^{32} - 2) + b + 2t$ shows that y is congruent to $b + 2t$ modulo $2^{32} - 2$, and that is the basis of our procedure. If the two-word sum $b + 2t$ does not extend into the top word and only requires one word, we have our new random number x; if it does extend into the top we must repeat the process. This may be described as a formal procedure, using top( ) and bot( ) to indicate the top and bottom halves of a two-word product or sum:

Form $x \leftarrow ax$ in adjoining 32-bit registers
Form $x \leftarrow \text{bot}(x) + 2\text{top}(x)$ until $\text{top}(x) = 0$

The sum $b + 2t$ should be formed as $b + t + t$, or perhaps by shifting t, to avoid multiplication.

A machine language implementation of this procedure will produce random 32-bit integers, with period $2^{31} - 2$, from any initial 32-bit integer (the seed) except 0 or $2^{31} - 1$. Most languages invoking the procedure will view the results as signed integers in the range -2^{31} to $2^{31}-1$ inclusive. Such an integer may be converted to a uniform number on $(-1,1)$ by multiplying by the real 2^{-31}, but more rapidly in machine language by shifting and inserting the appropriate exponent. Similar manipulations can produce a uniform real on (0,1).

Many simulation programs benefit from having signed random integers

or reals available directly, avoiding the cost of, say, converting a uniform U on (0,1) to a V on (−1,1) by means of $V = 2U - 1$. For example, the fastest methods for generating normal and other symmetric random variables may suffer a significant loss in average generation time if only positive, rather than half-positive, half-negative random numbers are available.

The case for modulus $2^{32} - 5$

The ability to provide, through machine-language manipulations of the top and bottom words of a two-word sum or product, rapid reductions for a modulus near 2^{32} naturally raises the question of using the prime closest to 2^{32}. That prime is $2^{32} - 5$. We can make reductions modulo $2^{32} - 5$ in a manner similar to that for $2^{32} - 2$, except that to the bottom we must add 5 times the top, the latter accomplished through a shift and an add.

The same arguments apply for the prime modulus $2^{32} - 5$. To implement a generator for that modulus we need a primitive root. A good one is 69070. Thus, for any 32-bit seed x_0, except those representing 0,1,2,3,4, and −5, the sequence $x_n = 69070x_{n-1} \bmod 2^{32} - 5$ will have period $2^{32} - 6$ and will produce random signed integers if it is implemented in a system that interprets integer bit patterns as least-absolute residues of 2^{32} and reduction modulo $2^{32} - 5$ is effected by means of a machine language routine that forms $b + 5t$ until the new t vanishes, as described. The six integers 0,1,2,3,4, and −5 cannot appear—a drawback of no significance in a set of 2,147,483,643 positive and 2,147,483,647 negative integers.

Double Precision Implementations

If speed is not important, one may program a congruential generator in double precision. In doing so, one should choose modulus and multiplier to satisfy as many of these criteria as possible: long period, few short-period subsequences, good lattice and good performance on statistical tests. Marsaglia [6] and Knuth [3] discuss the lattice structure of congruential generators and Knuth describes standard tests of randomness. Marsaglia [7] describes more stringent tests.

With a prime modulus p the period will be $p - 1$, so p should be made as large as possible, subject to the restriction that the primitive root a of p produce a generator with good lattice properties and the product ap not exceed 2^{53}, to avoid loss of bits in the generating process. One can usually find good multipliers a with from 14 to 20 bits, thus suggesting primes p of some 33 to 39 bits.

Periods of subsequences of congruential sequences depend on factors of $p - 1$, the period. As long as one is free to choose p, subject to a few restrictions, one should consider those for which $p - 1$ has few factors. It always has 2 as a factor, of course, and the fewest factors come from having $(p - 1)/2$ also a prime. Primes p such that $(p - 1)/2$ is also prime are called *safeprimes*. Here is a list of the safeprimes p closest to 2^i for $i = 31$ to 40, together with the least primitive root for that p:

$$2^{31} - 69,\ 6;\ 2^{32} + 91,\ 5;\ 2^{33} - 9,\ 5;\ 2^{34} + 79,\ 5;\ 2^{35} - 849,\ 7$$
$$2^{36} - 137,7;2^{37} - 45,5;2^{38} - 401,5;\ 2^{39} - 381,\ 5;\ 2^{40} + 437,\ 6$$

For any of these safeprimes, every odd power of the listed least primitive root is also a primitive root. This provides a plentiful supply of full-period multipliers for those seeking good lattices. Simple methods for determining the lattice structure are in [6].

Other Generators

Several other kinds of random number generators have been developed, partly to overcome the problem of the relatively short periods of congruential generators in 32-bit machines. Among them are combination generators, lagged-Fibonacci generators, congruential generators with multiple lags and a new kind of generator, subtract-with-borrow [9], that is currently considered one of the best of all. It has simple arithmetic (subtraction) and astonishingly long periods, typically 2^{500} to 2^{1800}. A popular description is in *Science News* Nov. 9, 1991, and implementations are available on computer networks. (Write me or send email: geo@stat.fsu.edu)

A recent survey article by Anderson [1] describes all but the last of these, as does [7], which also describes tests more stringent than the standard ones for evaluating generators. In addition, reference [8] addresses the question of an "industry standard" or "universal" generator and suggests one that has been widely adopted; its period is some 2^{144}.

Summary

Generators using modulus $2^{31} - 1$ are attractive only if they are implemented in machine language, but for such implementations the modulus $2^{32} - 2$ is preferable: it has the same period, provides simpler implementations and produces both positive and negative random numbers. The prime $2^{32} - 5$, closest to 2^{32}, also provides simple and fast machine language implementations, both positive and negative random integers and has a longer period.

For double precision or other extended precision implementations, the modulus $p = 2^{31} - 1$ is a poor choice: it is relatively small and $p - 1$ has far too many divisors. Larger primes provide longer periods. Better are larger safeprimes p among those listed. For them, $(p - 1)/2$ is also prime and thus half of the residues of p are primitive roots. Example: 16807 is a primitive root of the safeprime $2^{38} - 401$. The generator $x_n = 16807x_{n-1} \bmod 2^{38} - 401$ can be implemented in double precision with no loss of bits and has period 128 times as long as that of the proposed "industry standard." But anyone using a double precision implementation of such generators would be better served with one such as $x_n = 534059x_{n-1} - 4416x_{n-2} \bmod p = 2^{31} - 69$, which has period $p^2 - 1$, about 2^{62} or 5×10^{18}.

While these arguments raise questions about the suitability of the congruential generator $x_n = 16807x_{n-1} \bmod 2^{31} - 1$ as an industry standard, a more important question is: *Should any generator be designated a standard?*

There is no question of the need for precise rules for determining the

elements of a sequence of supposedly random numbers, to ensure that experiments may be verified or be capable of exact duplication in a wide variety of computers. Methods such as those in [8] are directed to that end.

But the idea of an industry standard smacks of stagnation and self-satisfaction. In using deterministic methods to simulate randomness we are all, as Von Neumann said "in a state of sin." Every deterministic scheme for producing randomness must have applications for which it gives bad results. Only the collective experience and imagination of developers and users of random number generators will lead to a better understanding of what those applications are. Only through encouraging—rather than stifling—experimentation can we, as Shakespeare urged, "plate sin with gold."

George Marsaglia
Department of Statistics
The Florida State University
Tallahassee, FL

ANOTHER TEST FOR RANDOMNESS

Although the need for tests of pseudo-random number generators (RNG's) has been debated [10], sometimes testing proves valuable.

One simple test is as follows: Let the minimum possible output of a given RNG be 1, and the maximum possible output be H.

For each of the seeds $1 \le s \le N$, let the first n values of the RNG sequence starting with seed s be denoted $r_{s1}, r_{s2}, \ldots, r_{sn}$. The basic question is: For what i is r_{si} the maximum of the sequence $r_{s1}, r_{s2}, \ldots, r_{sn}$?

Let $N \ll n \ll H$. Let $M_s = max(r_{si}: 1 \le i \le n)$, and let I_s be the lowest index i such that $r_{si} = M_s$. That is I_s is the index of the maximum value for seed s.

Since $n \ll H$, one would expect the I_s values to be distributed uniformly over the values $1 \le I_s \le n$. In fact, this is a classical "balls and urns" problem: each maximum corresponds to drawing, with replacement, one of a set of balls numbered 1 to n from an urn.

Since $N \ll n$, one would expect that no ball (or value of I_s) would be drawn more than a few times. The probability that at least one ball is drawn more than k times may be found as follows: Let $A_{k,i}$ be the event that ball i is drawn no more than k times. Let B_k be the event at least one ball is drawn more than k times.

$$P[B_k] = P[\neg A_1 \vee \neg A_2 \vee \ldots \vee \neg A_n] \le \Sigma_j P[\neg A_j] = nP[\neg A_1]$$

Now, $P[\neg A_1] = 1 - P[A_1] = \Sigma_{j=0}^{k} \binom{N}{j} p^j (1-p)^{N-j}$

where $\binom{N}{j}$ is the binomial coefficient $\frac{N!}{j!(N-j)!}$ and $p = 1/n$.

So $P[B_k] \le n[1 - \Sigma_{j=0}^{k} \binom{N}{j} p^j (1-p)^{N-j}]$

For $N = 100$ and $n = 10^5$, calculation yields $P[B_{10}] \le 2.35 \times 10^{-25}$.

In English: when using 100 different seeds, the probability that the sequence's maximum value occurs at the same index I_s in the sequence, for more than 10 different seeds, is $\le 2.35 \times 10^{-25}$.

Following are results of tests on several published RNGs, using $N = 100$ and $N = 10000$. Marsaglia's subtract-with-borrow generator [9]:

$x_n = x_{n-s} - x_{n-r} - c \ mod \ b$, using $b = 2^{31}$, $r = 48$, and $s = 8$: All values of I_s occurred exactly once. For results on Press's quick linear congruential methods [12] refer to Figure 1. For results on Park and Miller's linear congruential method refer to Figure 2. Note here that $I_s = 1311$ occurs 97 times.

The probability of any I_s occurring more than 96 times in a truly random sequence is less than 8.37×10^{-344}. Apparently there is a serious flaw in the Park-Miller RNG.

Stephen J. Sullivan
Mathcom Inc.
Lafayette, CO

Response

Our 1988 article in *Communications* [10] has proven to be often cited, in print and on electronic networks, a tribute in part to the widespread readership of *Communications*. There are four points we want to make relative to our article. We tried to stress a random-number-generator philosophy of simplicity, portability and efficiency. Consistent with this philosophy, we presented a particular example of a Lehmer generator and advocated it as a "minimal standard." To define the phrase minimal standard, in the third paragraph we wrote ". . . this is the generator that should always be used—unless one has access to a random number generator known to be better." At another point we wrote ". . . this represents a good minimal standard generator against which all other random number generators can—

Figure 1. Press's linear congruential methods

im	ia	ic	occurrences of I_s
134456	8121	28411	All values of I_s occurred exactly once.
714025	1366	150889	All values of I_s occurred exactly once.
259200	7141	54773	Two values of I_s occurred twice; all remaining ones occurred once.

Figure 2. Park and Miller's linear congruential method

I_s	number of occurrences of I_s
1260	1
1311	97
5230	1
6874	1

and should—be judged," and in the conclusion we wrote ". . . if you are not a specialist in random number generation and do not want to become one, use the minimal standard." We neither stated or implied that this generator should or would put to rest the endless quest by specialists for ever-better random number generators.

The minimal standard Lehmer generator we advocated had a modulus of $m = 2^{31} - 1$ and a multiplier of $a = 16807$. Relative to this particular choice of multiplier, we wrote ". . . if this paper were to be written again in a few years it is quite possible that we would advocate a different multiplier. . . ." We are now prepared to do so. That is, we now advocate $a = 48271$ and, indeed, have done so "officially" since July 1990. This new advocacy is consistent with the discussion on page 1198 of [10]. There is nothing wrong with 16807; we now believe, however, that 48271 is a little better (with $q = 44488$, $r = 3399$).

We have great respect for the contribution made by the text *Numerical Recipes* [5]. The no-nonsense, plain-speaking tone of this text combined with the general high quality of the software it contains, has made it a valuable reference. We say this despite the fact that the first edition had a relatively weak chapter on random number generation. We were pleased, therefore, to find that this chapter has been modified extensively in the recently published second edition, based in part on our article and recent work by Pierre L'Ecuyer.

In our article, we failed to mention an important feature that all good random number generators should have—the ability to generate multiple streams of random numbers. Space limitations prevent us from providing justification for this statement or discussing how nicely Lehmer generators are in general, and the minimal standard in particular, provide a multistream capability. For those interested, however, source code (in C or Pascal) for a 256-stream implementation of the $a = 48271$ generator is available. The file rngs.tar, a Unix tar archive of the source code, can be retrieved in the pub directory of the anonymous ftp account on ftp.cs.wm.edu (please use binary transfer mode).

Comments on Marsaglia

Marsaglia has a long history of significant contributions to the field of random number generation. We yield to his expertise relative to the mathematical theory of Lehmer generators and do not refute any of his math. We do, however, have a problem with his interpretation of our original article and his advocacy of register-level programming in this application.

We never proposed an "industry standard" random number generator. Indeed, we only used this phrase once, in the first paragraph of our original article, and then only in the context of characterizing the generally bad state of random number generators used in practice. If some self-appointed experts on the Internet have used our original article as a justification for stifling additional research, then we are disappointed that the system is not working correctly. If, instead Marsaglia thinks we are guilty of trying to stifle additional research, then we hope he will now recognize that wasn't our intent.

Independent of the statistical goodness of the generator advocated, we reject the register-level algorithm description and implementation Marsaglia recommends. We know from experience that this kind of thinking and programming leads to nonportable, obscure random-number-generation source code which violates good software engineering principles and practices. There are applications for which assembly language programming is necessary; this isn't one of them. Avoid the use of any generator that can't be clearly, efficiently and portably implemented in a high-level language.

To summarize, there are five principle points Marsaglia makes. We generally agree with the first, third, and fourth. We reject, however, the premise of the second point—there is no need or justification for a "tricky machine language implementation" of the minimal standard generator. Relative to the fifth point, we reiterate our advocacy of a minimal standard and our opposition to the more rigid, formal notion of an industry standard random number generator.

Comments on Sullivan

All random number generation algorithms are deterministic. Given that, one can always find statistical tests which even the best generators will fail, perhaps spectacularly. That is what Sullivan has done. We found the test results interesting and initially puzzling. One of us, however soon provided a mathematical explanation which solved the puzzle, making it clear that, to some extent, the "failure" is a characteristic of *all* Lehmer generators. Moreover, the "failure" is so spectacular because Sullivan uses the smallest possible initial seeds 1, 2, . . . N. That is, for example, if instead N initial 31-bit integer seeds are selected *at random* (using, say another generator or the state of the generator at the end of one test as the next initial seed) the "Park-Miller RNG" will pass Sullivan's test without difficulty.

The kind of weakness identified by Sullivan's test would be most likely to be a potential problem in replicated trials of a stochastic experiment. This problem would arise in this application, however, only if one seeded each trial with the index of the trial or in some other simple deterministic way. Reseeding in this way would be contrary to standard practice (which is to use the state of the generator at the end of one trial as the initial seed for the next trial), but we concede that some naive users might use the minimal standard generator in this way and, in that sense, Sullivan has identified a serious Lehmer generator flaw. Unfortunately, space limitations prohibit us from discussing this further.

The second edition of *Numerical Recipes* advocates the minimal standard generator in the sense we intended. The second edition also advocates an enhancement of the generator using a standard shuffling algorithm [12]. We generally endorse this enhancement. Indeed, we have verified that the ($a = 16807$ or $a =$

48271) minimal standard generator with shuffling passes Sullivan's test as well as the other two generators he cites (one of which is no longer advocated in *Numerical Recipes*).

Comments on Carta

In retrospect, when Carta's original article was first published [2], we should have commented on it. We didn't, however, so now is the time to do so. Although the article's title may suggest otherwise, the generator as implemented by Carta is *not* the minimal standard; it isn't even a full-period generator. We know of no good reason to use Carta's generator. Moreover, for the reasons mentioned previously, we reject the associated register-level programming he advocates as antithetic to our random number generation philosophy.

Summary

The construction of a simple, portable, efficient algorithm which can simulate randomness well enough to pass any reasonable statistical test remains a challenging activity today, just as it was more than 40 years ago when Lehmer's original paper was first published. The needs today, however, are more demanding. Computers can now easily run simulations which consume thousands of random numbers per second, thereby increasing the demand for generators with periods much larger than 2^{31}. Moreover, the increased use of parallel machines has generated a significant interest in parallel algorithms for random number generation. For all these reasons, random number generation remains an active, albeit highly specialized, area of research.

Given the dynamic nature of the area, it is difficult for nonspecialists to make decisions about what generator to use. "Give me something I can understand, implement and port . . . it needn't be state-of-the-art, just make sure it's reasonably good and efficient." Our article and the associated minimal standard generator was an attempt to respond to this request. Five years later, we see no need to alter our response other than to suggest the use of the multiplier $a = 48271$ in place of 16807.

Stephen K. Park
Keith W. Miller
Paul K. Stockmeyer
Dept. of Computer Science, The College of William & Mary
Williamsburg, VA

References

1. Anderson, S.L. Random number generators on vector supercomputers and other advanced architectures, *SIAM Review, 32*, 6 (June 1990), 221–251.
2. Carta, D.G. Two fast implementations of the "minimal standard" random number generator, *Commun. ACM 33*, 1(Jan. 1990), 87–88.
3. Knuth, D.E. *The Art of Computer Programming*, 2nd Ed. Addison Wesley, Reading, Mass. 1981.
4. Marsaglia, G. Random numbers fall mainly in the planes, *Proc. Nat. Acad. Sciences USA, 61* (1968), 25–28.
5. Marsaglia, G. and Bray, T. One-line random number generators and their use in combinations, *Commun. ACM, 11* (1968), 757–759.
6. Marsaglia, G. The structure of linear congruential sequences. *Applications of Number Theory to Numerical Analysis*, Z.K. Zaremba, Ed., Academic Press: New York (1972), 249–285.
7. Marsaglia, G. A current view of random number generators. Keynote Address, Computer Science and Statistics: 16th Symposium on the Interface. In *Proceedings of the Symposium*, L. Billard, Ed., (North-Holland, Amsterdam, 3–10).
8. Marsaglia, G., Zaman, A. and Tsang, W. Toward a universal random number generator. *Statistics and Probability Letters, 9* (Jan. 1990) 35–39.
9. Marsaglia, G. and Zaman, A. A new class of random number generators, *Annals of Applied Probability 1*, 3 (1991) 462–480.
10. Park, S.K. and Miller, K.W. Random number generators: Good ones are hard to find. *Commun. ACM 31*, 10 (Oct. 1988), 1192–1201.
11. 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.
12. Press, W.H., Flannery, B.P., Teukolsky, S.A. and Vetterling, W.T. *Numerical Recipes in C*. 2nd Edition, Cambridge University Press, Cambridge Mass., 1993. ◻