Random Numbers

CHAPTER 7

7.0 Introduction

It may seem perverse to use a computer, that most precise and deterministic of all machines conceived by the human mind, to produce "random" numbers. More than perverse, it may seem to be a conceptual impossibility. After all, any program produces output that is entirely predictable, hence not truly "random."

Nevertheless, practical computer "random number generators" are in common use. We will leave it to philosophers of the computer age to resolve the paradox in a deep way (see, e.g., Knuth [1] §3.5 for discussion and references). One sometimes hears computer-generated sequences termed *pseudo-random*, while the word *random* is reserved for the output of an intrinsically random physical process, like the elapsed time between clicks of a Geiger counter placed next to a sample of some radioactive element. We will not try to make such fine distinctions.

A working definition of randomness in the context of computer-generated sequences is to say that the deterministic program that produces a random sequence should be different from, and — in all measurable respects — statistically uncorrelated with, the computer program that *uses* its output. In other words, any two different random number generators ought to produce statistically the same results when coupled to your particular applications program. If they don't, then at least one of them is not (from your point of view) a good generator.

The above definition may seem circular, comparing, as it does, one generator to another. However, there exists a large body of random number generators that mutually do satisfy the definition over a very, very broad class of applications programs. And it is also found empirically that statistically identical results are obtained from random numbers produced by physical processes. So, because such generators are known to exist, we can leave to the philosophers the problem of defining them.

The pragmatic point of view is thus that randomness is in the eye of the beholder (or programmer). What is random enough for one application may not be random enough for another. Still, one is not entirely adrift in a sea of incommensurable applications programs: There is an accepted list of statistical tests, some sensible and some merely enshrined by history, that on the whole do a very good job of ferreting out any nonrandomness that is likely to be detected by an applications program (in this case, yours). Good random number generators ought to pass all of these tests,

or at least the user had better be aware of any that they fail, so that he or she will be able to judge whether they are relevant to the case at hand.

For references on this subject, the one to turn to first is Knuth [1]. Be cautious about any source earlier than about 1995, since the field progressed enormously in the following decade.

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1997, *Seminumerical Algorithms*, 3rd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), Chapter 3, especially §3.5.[1]

Gentle, J.E. 2003, *Random Number Generation and Monte Carlo Methods*, 2nd ed. (New York: Springer).

7.1 Uniform Deviates

Uniform deviates are just random numbers that lie within a specified range, typically 0.0 to 1.0 for floating-point numbers, or 0 to $2^{32} - 1$ or $2^{64} - 1$ for integers. Within the range, any one number is just as likely as any other. They are, in other words, what you probably think "random numbers" are. However, we want to distinguish uniform deviates from other sorts of random numbers, for example, numbers drawn from a normal (Gaussian) distribution of specified mean and standard deviation. These other sorts of deviates are almost always generated by performing appropriate operations on one or more uniform deviates, as we will see in subsequent sections. So, a reliable source of random uniform deviates, the subject of this section, is an essential building block for any sort of stochastic modeling or Monte Carlo computer work.

The state of the art for generating uniform deviates has advanced considerably in the last decade and now begins to resemble a mature field. It is now reasonable to expect to get "perfect" deviates in no more than a dozen or so arithmetic or logical operations per deviate, and fast, "good enough" deviates in many fewer operations than that. Three factors have all contributed to the field's advance: first, new mathematical algorithms; second, better understanding of the practical pitfalls; and, third, standardization of programming languages in general, and of integer arithmetic in particular — and especially the universal availability of unsigned 64-bit arithmetic in C and C++. It may seem ironic that something as down-in-the-weeds as this last factor can be so important. But, as we will see, it really is.

The greatest lurking danger for a user today is that many out-of-date and inferior methods remain in general use. Here are some traps to watch for:

- Never use a generator principally based on a *linear congruential generator* (LCG) or a *multiplicative linear congruential generator* (MLCG). We say more about this below.
- Never use a generator with a period less than $\sim 2^{64} \approx 2 \times 10^{19}$, or any generator whose period is undisclosed.
- Never use a generator that warns against using its low-order bits as being completely random. That was good advice once, but it now indicates an obsolete algorithm (usually a LCG).

- Never use the built-in generators in the C and C++ languages, especially `rand` and `srand`. These have no standard implementation and are often badly flawed.

If all scientific papers whose results are in doubt because of one or more of the above traps were to disappear from library shelves, there would be a gap on each shelf about as big as your fist.

You may also want to watch for indications that a generator is overengineered, and therefore wasteful of resources:

- Avoid generators that take more than (say) two dozen arithmetic or logical operations to generate a 64-bit integer or double precision floating result.
- Avoid using generators (over-)designed for serious cryptographic use.
- Avoid using generators with period $> 10^{100}$. You *really* will never need it, and, above some minimum bound, the period of a generator has little to do with its quality.

Since we have told you what to avoid from the past, we should immediately follow with the received wisdom of the present:

> An acceptable random generator must combine at least two (ideally, unrelated) methods. The methods combined should evolve independently and share no state. The combination should be by simple operations that do not produce results less random than their operands.

If you don't want to read the rest of this section, then use the following code to generate all the uniform deviates you'll ever need. This is our suspenders-and-belt, full-body-armor, never-any-doubt generator;* and, it also meets the above guidelines for avoiding wasteful, overengineered methods. (The fastest generators that we recommend, below, are only $\sim$2.5 $\times$ faster, even when their code is copied inline into an application.)

ran.h

```
struct Ran {
```

Implementation of the highest quality recommended generator. The constructor is called with an integer seed and creates an instance of the generator. The member functions `int64`, `doub`, and `int32` return the next values in the random sequence, as a variable type indicated by their names. The period of the generator is $\approx 3.138 \times 10^{57}$.

```
    Ullong u,v,w;
    Ran(Ullong j) : v(4101842887655102017LL), w(1) {
```

Constructor. Call with any integer seed (except value of `v` above).

```
        u = j ^ v; int64();
        v = u; int64();
        w = v; int64();
    }
    inline Ullong int64() {
```

*"What about the $1000 reward?" some long-time readers may wonder. That is a tale in itself: Two decades ago, the first edition of *Numerical Recipes* included a flawed random number generator. (Forgive us, we were young!) In the second edition, in a misguided attempt to buy back some credibility, we offered a prize of $1000 to the "first reader who convinces us" that that edition's best generator was in any way flawed. No one ever won that prize (`ran2` is a sound generator within its stated limits). We did learn, however, that many people don't understand what constitutes a statistical proof. Multiple claimants over the years have submitted claims based on one of two fallacies: (1) finding, after much searching, some particular seed that makes the first few random values seem unusual, or (2) finding, after some millions of trials, a statistic that, just that once, is as unlikely as a part in a million. In the interests of our own sanity, we are not offering any rewards in this edition. And the previous offer is hereby revoked.

```
    Return 64-bit random integer. See text for explanation of method.
        u = u * 2862933555777941757LL + 7046029254386353087LL;
        v ^= v >> 17; v ^= v << 31; v ^= v >> 8;
        w = 4294957665U*(w & 0xffffffff) + (w >> 32);
        Ullong x = u ^ (u << 21); x ^= x >> 35; x ^= x << 4;
        return (x + v) ^ w;
    }
    inline Doub doub() { return 5.42101086242752217E-20 * int64(); }
    Return random double-precision floating value in the range 0. to 1.
    inline Uint int32() { return (Uint)int64(); }
    Return 32-bit random integer.
};
```

The basic premise here is that a random generator, because it maintains internal state between calls, should be an object, a `struct`. You can declare more than one instance of it (although it is hard to think of a reason for doing so), and different instances will in no way interact.

The constructor `Ran()` takes a single integer argument, which becomes the seed for the sequence generated. Different seeds generate (for all practical purposes) completely different sequences. Once constructed, an instance of `Ran` offers several different formats for random output. To be specific, suppose you have created an instance by the declaration

```
Ran myran(17);
```

where `myran` is now the name of this instance, and 17 is its seed. Then, the function `myran.int64()` returns a random 64-bit unsigned integer; the function `myran.int32()` returns an unsigned 32-bit integer; and the function `myran.doub()` returns a double-precision floating value in the range 0.0 to 1.0. You can intermix calls to these functions as you wish. You can use *any* returned random bits for any purpose. If you need a random integer between 1 and `n` (inclusive), say, then the expression `1 + myran.int64() % (n-1)` is perfectly OK (though there are faster idioms than the use of %).

In the rest of this section, we briefly review some history (the rise and fall of the LCG), then give details on some of the algorithmic methods that go into a good generator, and on how to combine those methods. Finally, we will give some further recommended generators, additional to `Ran` above.

7.1.1 Some History

With hindsight, it seems clear that the whole field of random number generation was mesmerized, for far too long, by the simple recurrence equation

$$I_{j+1} = aI_j + c \pmod{m} \tag{7.1.1}$$

Here m is called the *modulus*, a is a positive integer called the *multiplier*, and c (which may be zero) is nonnegative integer called the *increment*. For $c \neq 0$, equation (7.1.1) is called a linear congruential generator (LCG). When $c = 0$, it is sometimes called a multiplicative LCG or MLCG.

The recurrence (7.1.1) must eventually repeat itself, with a period that is obviously no greater than m. If m, a, and c are properly chosen, then the period will be of maximal length, i.e., of length m. In that case, all possible integers between 0 and $m-1$ occur at some point, so any initial "seed" choice of I_0 is as good as any other:

The sequence just takes off from that point, and successive values I_j are the returned "random" values.

The idea of LCGs goes back to the dawn of computing, and they were widely used in the 1950s and thereafter. The trouble in paradise first began to be noticed in the mid-1960s (e.g., [1]): If k random numbers at a time are used to plot points in k-dimensional space (with each coordinate between 0 and 1), then the points will not tend to "fill up" the k-dimensional space, but rather will lie on $(k-1)$-dimensional "planes." There will be *at most* about $m^{1/k}$ such planes. If the constants m and a are not very carefully chosen, there will be *many fewer than that.* The number m was usually close to the machine's largest representable integer, often $\sim 2^{32}$. So, for example, the number of planes on which triples of points lie in three-dimensional space can be no greater than about the cube root of 2^{32}, about 1600. You might well be focusing attention on a physical process that occurs in a small fraction of the total volume, so that the discreteness of the planes can be very pronounced.

Even worse, many early generators happened to make particularly bad choices for m and a. One infamous such routine, `RANDU`, with $a = 65539$ and $m = 2^{31}$, was widespread on IBM mainframe computers for many years, and widely copied onto other systems. One of us recalls as a graduate student producing a "random" plot with only 11 planes and being told by his computer center's programming consultant that he had misused the random number generator: "We guarantee that each number is random individually, but we don't guarantee that more than one of them is random." That set back our graduate education by at least a year!

LCGs and MLCGs have additional weaknesses: When m is chosen as a power of 2 (e.g., `RANDU`), then the low-order bits generated are hardly random at all. In particular, the least significant bit has a period of at most 2, the second at most 4, the third at most 8, and so on. But, if you don't choose m as a power of 2 (in fact, choosing m prime is generally a good thing), then you generally need access to double-length registers to do the multiplication and modulo functions in equation (7.1.1). These were often unavailable in computers of the time (and usually still are).

A lot of effort subsequently went into "fixing" these weaknesses. An elegant number-theoretical test of m and a, the *spectral test*, was developed to characterize the density of planes in arbitrary dimensional space. (See [2] for a recent review that includes graphical renderings of some of the appallingly poor generators that were used historically, and also [3].) *Schrage's method* [4] was invented to do the multiplication $a\,I_j$ with only 32-bit arithmetic for m as large as $2^{32}-1$, but, unfortunately, only for certain a's, not always the best ones. The review by Park and Miller [5] gives a good contemporary picture of LCGs in their heyday.

Looking back, it seems clear that the field's long preoccupation with LCGs was somewhat misguided. There is no technological reason that the better, non-LCG, generators of the last decade could not have been discovered decades earlier, nor any reason that the impossible dream of an elegant "single algorithm" generator could not also have been abandoned much earlier (in favor of the more pragmatic patchwork in combined generators). As we will explain below, LCGs and MLCGs can still be useful, but only in carefully controlled situations, and with due attention to their manifest weaknesses.

7.1.2 Recommended Methods for Use in Combined Generators

Today, there are at least a dozen plausible algorithms that deserve serious consideration for use in random generators. Our selection of a few is motivated by aesthetics as much as mathematics. We like algorithms with few and fast operations, with foolproof initialization, and with state small enough to keep in registers or first-level cache (if the compiler and hardware are able to do so). This means that we tend to avoid otherwise fine algorithms whose state is an array of some length, despite the relative simplicity with which such algorithms can achieve truly humongous periods. For overviews of broader sets of methods, see [6] and [7].

To be recommendable for use in a combined generator, we require a method to be understood theoretically to some degree, and to pass a reasonably broad suite of empirical tests (or, if it fails, have weaknesses that are well characterized). Our minimal theoretical standard is that the period, the set of returned values, and the set of valid initializations should be completely understood. As a minimal empirical standard, we have used the second release (2003) of Marsaglia's whimsically named Diehard battery of statistical tests [8].* An alternative test suite, NIST-STS [9], might be used instead, or in addition.

Simply requiring a combined generator to pass Diehard or NIST-STS is not an acceptably stringent test. These suites make only $\sim 10^7$ calls to the generator, whereas a user program might make 10^{12} or more. Much more meaningful is to require that each method in a combined generator separately pass the chosen suite. Then the combination generator (if correctly constructed) should be vastly better than any one component. In the tables below, we use the symbol "$*$" to indicate that a method passes the Diehard tests by itself. (For 64-bit quantities, the statement is that the 32 high and low bits each pass.) Correspondingly, the words "can be used as random," below, do not imply perfect randomness, but only a minimum level for quick-and-dirty applications where a better, combined, generator is just not needed.

We turn now to specific methods, starting with methods that use 64-bit unsigned arithmetic (what we call `Ullong`, that is, `unsigned long long` in the Linux/Unix world, or `unsigned __int64` on planet Microsoft).

(A) 64-bit Xorshift Method. This generator was discovered and characterized by Marsaglia [10]. In just three XORs and three shifts (generally fast operations) it produces a full period of $2^{64} - 1$ on 64 bits. (The missing value is zero, which perpetuates itself and must be avoided.) High and low bits pass Diehard. A generator can use either the three-line update rule, below, that starts with <<, or the rule that starts with >>. (The two update rules produce different sequences, related by bit reversal.)

state:	x (unsigned 64-bit)
initialize:	$x \neq 0$
update:	$x \leftarrow x \wedge (x \ >> \ a_1)$,
	$x \leftarrow x \wedge (x \ << \ a_2)$,
	$x \leftarrow x \wedge (x \ >> \ a_3)$;
or	$x \leftarrow x \wedge (x \ << \ a_1)$,
	$x \leftarrow x \wedge (x \ >> \ a_2)$,

*Be sure that you use a version of Diehard that includes the so-called "Gorilla Test."

	$x \leftarrow x \wedge (x \ll a_3)$;
can use as random:	x (all bits) ⁂
can use in bit mix:	x (all bits)
can improve by:	output 64-bit MLCG successor
period:	$2^{64} - 1$

Here is a very brief outline of the theory behind these generators: Consider the 64 bits of the integer as components in a vector of length 64, in a linear space where addition and multiplication are done modulo 2. Noting that XOR ($\wedge$) is the same as addition, each of the three lines in the updating can be written as the action of a 64×64 matrix on a vector, where the matrix is all zeros except for ones on the diagonal, and on exactly one super- or subdiagonal (corresponding to << or >>). Denote this matrix as $\mathbf{S}_k$, where k is the shift argument (positive for left-shift, say, and negative for right-shift). Then, one full step of updating (three lines of the updating rule, above) corresponds to multiplication by the matrix $\mathbf{T} \equiv \mathbf{S}_{k_3}\mathbf{S}_{k_2}\mathbf{S}_{k_1}$.

One next needs to find triples of integers (k_1, k_2, k_3), for example $(21, -35, 4)$, that give the full $M \equiv 2^{64} - 1$ period. Necessary and sufficient conditions are that $\mathbf{T}^M = \mathbf{1}$ (the identity matrix) and that $\mathbf{T}^N \neq \mathbf{1}$ for these seven values of N: $M/6700417$, $M/65537$, $M/641$, $M/257$, $M/17$, $M/5$, and $M/3$, that is, M divided by each of its seven distinct prime factors. The required large powers of $\mathbf{T}$ are readily computed by successive squarings, requiring only on the order of 64^4 operations. With this machinery, one can find full-period triples (k_1, k_2, k_3) by exhaustive search, at a reasonable cost.

Brent [11] has pointed out that the 64-bit xorshift method produces, at each bit position, a sequence of bits that is identical to one produced by a certain linear feedback shift register (LFSR) on 64 bits. (We will learn more about LFSRs in §7.5.) The xorshift method thus potentially has some of the same strengths and weaknesses as an LFSR. Mitigating this, however, is the fact that the primitive polynomial equivalent of a typical xorshift generator has many nonzero terms, giving it better statistical properties than LFSR generators based, for example, on primitive trinomials. In effect, the xorshift generator is a way to step simultaneously 64 nontrivial one-bit LFSR registers, using only six fast, 64-bit operations. There are other ways of making fast steps on LFSRs, and combining the output of more than one such generator [12,13], but none as simple as the xorshift method.

While each bit position in an xorshift generator has the same recurrence, and therefore the same sequence with period $2^{64} - 1$, the method guarantees offsets to each sequence such that all nonzero 64-bit words are produced *across* the bit positions during one complete cycle (as we just saw).

A selection of full-period triples is tabulated in [10]. Only a small fraction of full-period triples actually produce generators that pass Diehard. Also, a triple may pass in its <<-first version, and fail in its >>-first version, or vice versa. Since the two versions produce simply bit-reversed sequences, a failure of either sense must obviously be considered a failure of both (and a weakness in Diehard). The following recommended parameter sets pass Diehard for both the << and >> rules. The sets near the top of the list may be slightly superior to the sets near the bottom. The column labeled ID assigns an identification string to each recommended generator that we will refer to later.

ID	a_1	a_2	a_3
A1	21	35	4
A2	20	41	5
A3	17	31	8
A4	11	29	14
A5	14	29	11
A6	30	35	13
A7	21	37	4
A8	21	43	4
A9	23	41	18

It is easy to design a test that the xorshift generator fails if used by itself. Each bit at step $i + 1$ depends on at most 8 bits of step i, so some simple logical combinations of the two timesteps (and appropriate masks) will show immediate nonrandomness. Also, when the state passes though a value with only small numbers of 1 bits, as it must eventually do (so-called states of *low Hamming weight*), it will take longer than expected to recover. Nevertheless, used in combination, the xorshift generator is an exceptionally powerful and useful method. Much grief could have been avoided had it, instead of LCGs, been discovered in 1949!

(B) Multiply with Carry (MWC) with Base $b = 2^{32}$. Also discovered by Marsaglia, the *base* b of an MWC generator is most conveniently chosen to be a power of 2 that is half the available word length (i.e., $b = 32$ for 64-bit words). The MWC is then defined by its *multiplier* a.

state:	x (unsigned 64-bit)
initialize:	$1 \le x \le 2^{32} - 1$
update:	$x \leftarrow a\ (x\ \&\ [2^{32} - 1]) + (x\ >>\ 32)$
can use as random:	x (low 32 bits) ✳
can use in bit mix:	x (all 64 bits)
can improve by:	output 64-bit xorshift successor to 64 bit x
period:	$(2^{32}a - 2)/2$ (a prime)

An MWC generator with parameters b and a is related theoretically [14] to, though not identical to, an LCG with modulus $m = ab-1$ and multiplier a. It is easy to find values of a that make m a prime, so we get, in effect, the benefit of a prime modulus using only power-of-two modular arithmetic. It is not possible to choose a to give the maximal period m, but if a is chosen to make both m and $(m - 1)/2$ prime, then the period of the MCG is $(m - 1)/2$, almost as good. A fraction of candidate a's thus chosen passes the standard statistical test suites; a spectral test [14] is a promising development, but we have not made use of it here.

Although only the low b bits of the state x can be taken as algorithmically random, there is considerable randomness in all the bits of x that represent the product ab. This is very convenient in a combined generator, allowing the entire state x to be used as a component. In fact, the first two recommended a's below give ab so close to 2^{64} (within about 2 ppm) that the high bits of x actually pass Diehard. (This is a good example of how any test suite can fail to find small amounts of highly nonrandom behavior, in this case as many as 8000 missing values in the top 32 bits.)

Apart from this kind of consideration, the values below are recommended with no particular ordering.

ID	a
B1	4294957665
B2	4294963023
B3	4162943475
B4	3947008974
B5	3874257210
B6	2936881968
B7	2811536238
B8	2654432763
B9	1640531364

(C) LCG Modulo 2^{64}. Why in the world do we include this generator after vilifying it so thoroughly above? For the parameters given (which strongly pass the spectral test), its high 32 bits almost, but don't quite, pass Diehard, and its low 32 bits are a complete disaster. Yet, as we will see when we discuss the construction of combined generators, there is still a niche for it to fill. The recommended multipliers a below have good spectral characteristics [15].

state:	x (unsigned 64-bit)
initialize:	any value
update:	$x \leftarrow ax + c \pmod{2^{64}}$
can use as random:	x (high 32 bits, with caution)
can use in bit mix:	x (high 32 bits)
can improve by:	output 64-bit xorshift successor
period:	2^{64}

ID	a	c (any odd value ok)
C1	3935559000370003845	2691343689449507681
C2	3202034522624059733	4354685564936845319
C3	2862933555777941757	7046029254386353087

(D) MLCG Modulo 2^{64}. As for the preceding one, the useful role for this generator is strictly limited. The low bits are highly nonrandom. The recommended multipliers have good spectral characteristics (some from [15]).

state:	x (unsigned 64-bit)
initialize:	$x \neq 0$
update:	$x \leftarrow ax \pmod{2^{64}}$
can use as random:	x (high 32 bits, with caution)
can use in bit mix:	x (high 32 bits)
can improve by:	output 64-bit xorshift successor
period:	2^{62}

ID	a
D1	2685821657736338717
D2	7664345821815920749
D3	4768777513237032717
D4	1181783497276652981
D5	702098784532940405

(E) MLCG with $m \gg 2^{32}$, m Prime. When 64-bit unsigned arithmetic is available, the MLCGs with prime moduli and large multipliers of good spectral character are decent 32-bit generators. Their main liability is that the 64-bit multiply and 64-bit remainder operations are quite expensive for the mere 32 (or so) bits of the result.

state:	x (unsigned 64-bit)
initialize:	$1 \le x \le m-1$
update:	$x \leftarrow ax \pmod m$
can use as random:	$x \quad (1 \le x \le m-1)$ or low 32 bits ✳
can use in bit mix:	(same)
period:	$m-1$

The parameter values below were kindly computed for us by P. L'Ecuyer. The multipliers are about the best that can be obtained for the prime moduli, close to powers of 2, shown. Although the recommended use is for only the low 32 bits (which all pass Diehard), you can see that (depending on the modulus) as many as 43 reasonably good bits can be obtained for the cost of the 64-bit multiply and remainder operations.

ID	m	a
E1	$2^{39}-7 = 549755813881$	10014146
E2		30508823
E3		25708129
E4	$2^{41}-21 = 2199023255531$	5183781
E5		1070739
E6		6639568
E7	$2^{42}-11 = 4398046511093$	1781978
E8		2114307
E9		1542852
E10	$2^{43}-57 = 8796093022151$	2096259
E11		2052163
E12		2006881

(F) MLCG with $m \gg 2^{32}$, m Prime, and $a(m-1) \approx 2^{64}$. A variant, for use in combined generators, is to choose m and a to make $a(m-1)$ as close as possible to 2^{64}, while still requiring that m be prime and that a pass the spectral test. The purpose of this maneuver is to make ax a 64-bit value with good randomness in its high bits, for use in combined generators. The expense of the multiply and remainder operations is still the big liability, however. The low 32 bits of x are not significantly less random than those of the previous MLCG generators E1–E12.

state:	x (unsigned 64-bit)
initialize:	$1 \le x \le m-1$
update:	$x \leftarrow ax \pmod m$
can use as random:	$x \quad (1 \le x \le m-1)$ or low 32 bits ⁂
can use in bit mix:	ax (but don't use both ax and x) ⁂
can improve by:	output 64-bit xorshift successor of ax
period:	$m-1$

ID	m	a
F1	$1148 \times 2^{32} + 11 = 4930622455819$	3741260
F2	$1264 \times 2^{32} + 9 = 5428838662153$	3397916
F3	$2039 \times 2^{32} + 3 = 8757438316547$	2106408

7.1.3 How to Construct Combined Generators

While the construction of combined generators is an art, it should be informed by underlying mathematics. Rigorous theorems about combined generators are usually possible only when the generators being combined are algorithmically related; but that in itself is usually a bad thing to do, on the general principle of "don't put all your eggs in one basket." So, one is left with guidelines and rules of thumb.

The methods being combined should be independent of one another. They must share no state (although their initializations are allowed to derive from some convenient common seed). They should have different, incommensurate, periods. And, ideally, they should "look like" each other algorithmically as little as possible. This latter criterion is where some art necessarily enters.

The output of the combination generator should in no way perturb the independent evolution of the individual methods, nor should the operations effecting combination have any side effects.

The methods should be combined by binary operations whose output is no less random than one input if the other input is held fixed. For 32- or 64-bit unsigned arithmetic, this in practice means that only the $+$ and $\wedge$ operators can be used. As an example of a forbidden operator, consider multiplication: If one operand is a power of 2, then the product will end in trailing zeros, no matter how random is the other operand.

All bit positions in the combined output should depend on high-quality bits from at least two methods, and may also depend on lower-quality bits from additional methods. In the tables above, the bits labeled "can use as random" are considered high quality; those labeled "can use in bit mix" are considered low quality, unless they also pass a statistical suite such as Diehard.

There is one further trick at our disposal, the idea of using a method as a *successor relation* instead of as a generator in its own right. Each of the methods described above is a mapping from some 64-bit state x_i to a unique successor state x_{i+1}. For a method to pass a good statistical test suite, it must have no detectable correlations between a state and its successor. If, in addition, the method has period 2^{64} or $2^{64}-1$, then all values (except possibly zero) occur exactly once as successor states.

Suppose we take the output of a generator, say C1 above, with period 2^{64}, and run it through generator A6, whose period is $2^{64}-1$, as a successor relation. This is conveniently denoted by "A6(C1)," which we will call a *composed* generator. Note that the composed output is emphatically *not* fed back into the state of C1, which

continues unperturbed. The composed generator A6(C1) has the period of C1, not, unfortunately, the product of the two periods. But its random mapping of C1's output values effectively fixes C1's problems with short-period low bits. (The better so if the form of A6 with left-shift first is used.) And, A6(C1) will also fix A6's weakness that a bit depends only on a few bits of the previous state. We will thus consider a carefully constructed composed generator as being a combined generator, on a par with direct combining via $+$ or $\wedge$.

Composition is inferior to direct combining in that it costs almost as much but does not increase the size of the state or the length of the period. It is superior to direct combining in its ability to mix widely differing bit positions. In the previous example we would not have accepted A6+C1 as a combined generator, because the low bits of C1 are so poor as to add little value to the combination; but A6(C1) has no such liability, and much to recommend it. In the preceding summary tables of each method, we have indicated recommended combinations for composed generators in the table entries, "can improve by."

We can now completely describe the generator in `Ran`, above, by the pseudo-equation,

$$\texttt{Ran} = [\mathrm{A1}_l(\mathrm{C3}) + \mathrm{A3}_r] \wedge \mathrm{B1} \tag{7.1.2}$$

that is, the combination and/or composition of four different generators. For the methods A1 and A3, the subscripts l and r denote whether a left- or right-shift operation is done first. The period of `Ran` is the least common multiple of the periods of C3, A3, and B1.

The simplest and fastest generator that we can readily recommend is

$$\texttt{Ranq1} \equiv \mathrm{D1}(\mathrm{A1}_r) \tag{7.1.3}$$

implemented as

ran.h

Recommended generator for everyday use. The period is $\approx 1.8 \times 10^{19}$. Calling conventions same as `Ran`, above.

```
struct Ranq1 {
    Ullong v;
    Ranq1(Ullong j) : v(4101842887655102017LL) {
        v ^= j;
        v = int64();
    }
    inline Ullong int64() {
        v ^= v >> 21; v ^= v << 35; v ^= v >> 4;
        return v * 2685821657736338717LL;
    }
    inline Doub doub() { return 5.42101086242752217E-20 * int64(); }
    inline Uint int32() { return (Uint)int64(); }
};
```

`Ranq1` generates a 64-bit random integer in 3 shifts, 3 xors, and one multiply, or a double floating value in one additional multiply. Its method is concise enough to go easily inline in an application. It has a period of "only" 1.8×10^{19}, so it should not be used by an application that makes more than $\sim 10^{12}$ calls. With that restriction, we think that `Ranq1` will do just fine for 99.99% of all user applications, and that `Ran` can be reserved for the remaining 0.01%.

If the "short" period of `Ranq1` bothers you (which it shouldn't), you can instead use

$$\texttt{Ranq2} \equiv \mathrm{A3}_r \wedge \mathrm{B1} \tag{7.1.4}$$

whose period is 8.5×10^{37}.

ran.h
```
struct Ranq2 {
```
Backup generator if Ranq1 has too short a period and Ran is too slow. The period is $\approx 8.5 \times 10^{37}$. Calling conventions same as Ran, above.
```
    Ullong v,w;
    Ranq2(Ullong j) : v(4101842887655102017LL), w(1) {
        v ^= j;
        w = int64();
        v = int64();
    }
    inline Ullong int64() {
        v ^= v >> 17; v ^= v << 31; v ^= v >> 8;
        w = 4294957665U*(w & 0xffffffff) + (w >> 32);
        return v ^ w;
    }
    inline Doub doub() { return 5.42101086242752217E-20 * int64(); }
    inline Uint int32() { return (Uint)int64(); }
};
```

7.1.4 Random Hashes and Random Bytes

Every once in a while, you want a random sequence H_i whose values you can visit or revisit in any order of i's. That is to say, you want a *random hash* of the integers i, one that passes serious tests for randomness, even for very ordered sequences of i's. In the language already developed, you want a generator that has no state at all and is built entirely of successor relationships, starting with the value i.

An example that easily passes the Diehard test is

$$\texttt{Ranhash} \equiv \mathrm{A2}_l(\mathrm{D3}(\mathrm{A7}_r(\mathrm{C1}(i)))) \tag{7.1.5}$$

Note the alternation between successor relations that utilize 64-bit multiplication and ones using shifts and XORs.

ran.h
```
struct Ranhash {
```
High-quality random hash of an integer into several numeric types.
```
    inline Ullong int64(Ullong u) {
    Returns hash of u as a 64-bit integer.
        Ullong v = u * 3935559000370003845LL + 2691343689449507681LL;
        v ^= v >> 21; v ^= v << 37; v ^= v >> 4;
        v *= 4768777513237032717LL;
        v ^= v << 20; v ^= v >> 41; v ^= v << 5;
        return  v;
    }
    inline Uint int32(Ullong u)
    Returns hash of u as a 32-bit integer.
        { return (Uint)(int64(u) & 0xffffffff) ; }
    inline Doub doub(Ullong u)
    Returns hash of u as a double-precision floating value between 0. and 1.
        { return 5.42101086242752217E-20 * int64(u); }
};
```

Since `Ranhash` has no state, it has no constructor. You just call its `int64(`i`)` function, or any of its other functions, with your value of i whenever you want.

Random Bytes. In a different set of circumstances, you may want to generate random integers a byte at a time. You can of course pull bytes out of any of the above

recommended combination generators, since they are constructed to be equally good on all bits. The following code, added to any of the generators above, augments them with an `int8()` method. (Be sure to initialize `bc` to zero in the constructor.)

```
Ullong breg;
Int bc;
inline unsigned char int8() {
    if (bc--) return (unsigned char)(breg >>= 8);
    breg = int64();
    bc = 7;
    return (unsigned char)breg;
}
```

If you want a more byte-oriented, though not necessarily faster, algorithm, an interesting one — in part because of its interesting history — is Rivest's RC4, used in many Internet applications. RC4 was originally a proprietary algorithm of RSA, Inc., but it was protected simply as a trade secret and not by either patent or copyright. The result was that when the secret was breached, by an anonymous posting to the Internet in 1994, RC4 became, in almost all respects, public property. The name RC4 is still protectable, and is a trademark of RSA. So, to be scrupulous, we give the following implementation another name, `Ranbyte`.

ran.h

```
struct Ranbyte {
Generator for random bytes using the algorithm generally known as RC4.
    Int s[256],i,j,ss;
    Uint v;
    Ranbyte(Int u) {
    Constructor. Call with any integer seed.
        v = 2244614371U ^ u;
        for (i=0; i<256; i++) {s[i] = i;}
        for (j=0, i=0; i<256; i++) {
            ss = s[i];
            j = (j + ss + (v >> 24)) & 0xff;
            s[i] = s[j]; s[j] = ss;
            v = (v << 24) | (v >> 8);
        }
        i = j = 0;
        for (Int k=0; k<256; k++) int8();
    }
    inline unsigned char int8() {
    Returns next random byte in the sequence.
        i = (i+1) & 0xff;
        ss = s[i];
        j = (j+ss) & 0xff;
        s[i] = s[j]; s[j] = ss;
        return (unsigned char)(s[(s[i]+s[j]) & 0xff]);
    }
    Uint int32() {
    Returns a random 32-bit integer constructed from 4 random bytes. Slow!
        v = 0;
        for (int k=0; k<4; k++) {
            i = (i+1) & 0xff;
            ss = s[i];
            j = (j+ss) & 0xff;
            s[i] = s[j]; s[j] = ss;
            v = (v << 8) | s[(s[i]+s[j]) & 0xff];
        }
        return v;
    }
```

```
    Doub doub() {
    Returns a random double-precision floating value between 0. and 1. Slow!!
        return 2.32830643653869629E-10 * ( int32() +
               2.32830643653869629E-10 * int32() );
    }
};
```

Notice that there is a lot of overhead in starting up an instance of `Ranbyte`, so you should not create instances inside loops that are executed many times. The methods that return 32-bit integers, or double floating-point values, are *slow* in comparison to the other generators above, but are provided in case you want to use `Ranbyte` as a test substitute for another, perhaps questionable, generator.

If you find any nonrandomness at all in `Ranbyte`, don't tell us. But there are several national cryptological agencies that might, or might not, want to talk to you!

7.1.5 Faster Floating-Point Values

The steps above that convert a 64-bit integer to a double-precision floating-point value involves both a nontrivial type conversion and a 64-bit floating multiply. They are performance bottlenecks. One can instead directly move the random bits into the right place in the `double` word with `union` structure, a mask, and some 64-bit logical operations; but in our experience this is not significantly faster.

To generate faster floating-point values, if that is an absolute requirement, we need to bend some of our design rules. Here is a variant of "Knuth's subtractive generator," which is a so-called *lagged Fibonacci generator* on a circular list of 55 values, with lags 24 and 55. Its interesting feature is that new values are generated directly as floating point, by the floating-point subtraction of two previous values.

ran.h
```
struct Ranfib {
Implements Knuth's subtractive generator using only floating operations. See text for cautions.
    Doub dtab[55], dd;
    Int inext, inextp;
    Ranfib(Ullong j) : inext(0), inextp(31) {
    Constructor. Call with any integer seed. Uses Ranq1 to initialize.
        Ranq1 init(j);
        for (int k=0; k<55; k++) dtab[k] = init.doub();
    }
    Doub doub() {
    Returns random double-precision floating value between 0. and 1.
        if (++inext == 55) inext = 0;
        if (++inextp == 55) inextp = 0;
        dd = dtab[inext] - dtab[inextp];
        if (dd < 0) dd += 1.0;
        return (dtab[inext] = dd);
    }
    inline unsigned long int32()
    Returns random 32-bit integer. Recommended only for testing purposes.
        { return (unsigned long)(doub() * 4294967295.0);}
};
```

The `int32` method is included merely for testing, or incidental use. Note also that we use `Ranq1` to initialize `Ranfib`'s table of 55 random values. See earlier editions of Knuth or *Numerical Recipes* for a (somewhat awkward) way to do the initialization purely internally.

`Ranfib` fails the Diehard "birthday test," which is able to discern the simple relation among the three values at lags 0, 24, and 55. Aside from that, it is a good,

but not great, generator, with speed as its principal recommendation.

7.1.6 Timing Results

Timings depend so intimately on highly specific hardware and compiler details, that it is hard to know whether a single set of tests is of any use at all. This is especially true of combined generators, because a good compiler, or a CPU with sophisticated instruction look-ahead, can interleave and pipeline the operations of the individual methods, up to the final combination operations. Also, as we write, desktop computers are in transition from 32 bits to 64, which will affect the timing of 64-bit operations. So, you ought to familiarize yourself with C's "`clock_t clock(void)`" facility and run your own experiments.

That said, the following tables give typical results for routines in this section, normalized to a 3.4 GHz Pentium CPU, vintage 2004. The units are 10^6 returned values per second. Large numbers are better.

Generator	`int64()`	`doub()`	`int8()`
`Ran`	19	10	51
`Ranq1`	39	13	59
`Ranq2`	32	12	58
`Ranfib`		24	
`Ranbyte`			43

The `int8()` timings for `Ran`, `Ranq1`, and `Ranq2` refer to versions augmented as indicated above.

7.1.7 When You Have Only 32-Bit Arithmetic

Our best advice is: Get a better compiler! But if you seriously must live in a world with only unsigned 32-bit arithmetic, then here are some options. None of these individually pass Diehard.

(G) 32-Bit Xorshift RNG

state:	x (unsigned 32-bit)
initialize:	$x \neq 0$
update:	$x \leftarrow x \wedge (x \gg b_1)$,
	$x \leftarrow x \wedge (x \ll b_2)$,
	$x \leftarrow x \wedge (x \gg b_3)$;
or	$x \leftarrow x \wedge (x \ll b_1)$,
	$x \leftarrow x \wedge (x \gg b_2)$,
	$x \leftarrow x \wedge (x \ll b_3)$;
can use as random:	x (32 bits, with caution)
can use in bit mix:	x (32 bits)
can improve by:	output 32-bit MLCG successor
period:	$2^{32} - 1$