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:	$x \leftarrow x \wedge (x \ \texttt{>>}\ a_1)$,
	$x \leftarrow x \wedge (x \ \texttt{<<}\ a_2)$,
	$x \leftarrow x \wedge (x \ \texttt{>>}\ a_3)$;
or	$x \leftarrow x \wedge (x \ \texttt{<<}\ a_1)$,
	$x \leftarrow x \wedge (x \ \texttt{>>}\ a_2)$,

*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

```
struct Ranq1 {
```
Recommended generator for everyday use. The period is $\approx 1.8 \times 10^{19}$. Calling conventions same as `Ran`, above.
```
    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:	$x \leftarrow x \wedge (x \texttt{ >> } b_1)$,
	$x \leftarrow x \wedge (x \texttt{ << } b_2)$,
	$x \leftarrow x \wedge (x \texttt{ >> } b_3)$;
or	$x \leftarrow x \wedge (x \texttt{ << } b_1)$,
	$x \leftarrow x \wedge (x \texttt{ >> } b_2)$,
	$x \leftarrow x \wedge (x \texttt{ << } b_3)$;
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:	$2^{32} - 1$

ID	b_1	b_2	b_3
G1	13	17	5
G2	7	13	3
G3	9	17	6
G4	6	13	5
G5	9	21	2
G6	17	15	5
G7	3	13	7
G8	5	13	6
G9	12	21	5

(H) MWC with Base $b = 2^{16}$

state:	x, y (unsigned 32-bit)
initialize:	$1 \le x, y \le 2^{16} - 1$
update:	$x \leftarrow a\ (x\ \&\ [2^{16} - 1]) + (x\ >>\ 16)$
	$y \leftarrow b\ (y\ \&\ [2^{16} - 1]) + (y\ >>\ 16)$
can use as random:	$(x\ <<\ 16) + y$
can use in bit mix:	same, or (with caution) x or y
can improve by:	output 32-bit xorshift successor
period:	$(2^{16}a - 2)(2^{16}b - 2)/4$ (product of two primes)

ID	a	b
H1	62904	41874
H2	64545	34653
H3	34653	64545
H4	57780	55809
H5	48393	57225
H6	63273	33378

(I) LCG Modulo 2^{32}

state:	x (unsigned 32-bit)
initialize:	any value
update:	$x \leftarrow ax + c \pmod{2^{32}}$
can use as random:	not recommended
can use in bit mix:	not recommended
can improve by:	output 32-bit xorshift successor
period:	2^{32}

ID	a	c (any odd ok)
I1	1372383749	1289706101
I2	2891336453	1640531513
I3	2024337845	797082193
I4	32310901	626627237
I5	29943829	1013904223

(J) MLCG Modulo 2^{32}

state:	x (unsigned 32-bit)
initialize:	$x \neq 0$
update:	$x \leftarrow ax \pmod{2^{32}}$
can use as random:	not recommended
can use in bit mix:	not recommended
can improve by:	output 32-bit xorshift successor
period:	2^{30}

ID	a
J1	1597334677
J2	741103597
J3	1914874293
J4	990303917
J5	747796405

A high-quality, if somewhat slow, combined generator is

$$\texttt{Ranlim32} \equiv [\mathrm{G3}_l(\mathrm{I2}) + \mathrm{G1}_r] \wedge [\mathrm{G6}_l(\mathrm{H6}_b) + \mathrm{H5}_b] \tag{7.1.6}$$

implemented as

ran.h

```
struct Ranlim32 {
```
High-quality random generator using only 32-bit arithmetic. Same conventions as `Ran`. Period $\approx 3.11 \times 10^{37}$. Recommended only when 64-bit arithmetic is not available.
```
    Uint u,v,w1,w2;
    Ranlim32(Uint j) : v(2244614371U), w1(521288629U), w2(362436069U) {
        u = j ^ v; int32();
        v = u; int32();
    }
    inline Uint int32() {
        u = u * 2891336453U + 1640531513U;
        v ^= v >> 13; v ^= v << 17; v ^= v >> 5;
        w1 = 33378 * (w1 & 0xffff) + (w1 >> 16);
        w2 = 57225 * (w2 & 0xffff) + (w2 >> 16);
        Uint x = u ^ (u << 9); x ^= x >> 17; x ^= x << 6;
        Uint y = w1 ^ (w1 << 17); y ^= y >> 15; y ^= y << 5;
        return (x + v) ^ (y + w2);
    }
    inline Doub doub() { return 2.3283064653869629E-10 * int32(); }
    inline Doub truedoub() {
        return 2.3283064653869629E-10 * ( int32() +
        2.3283064653869629E-10 * int32() );
    }
};
```

Note that the `doub()` method returns floating-point numbers with only 32 bits of precision. For full precision, use the slower `truedoub()` method.

CITED REFERENCES AND FURTHER READING:

Gentle, J.E. 2003, *Random Number Generation and Monte Carlo Methods*, 2nd ed. (New York: Springer), Chapter 1.

Marsaglia, G 1968, "Random Numbers Fall Mainly in the Planes", *Proceedings of the National Academy of Sciences*, vol. 61, pp. 25–28.[1]

Entacher, K. 1997, "A Collection of Selected Pseudorandom Number Generators with Linear Structures", Technical Report No. 97, Austrian Center for Parallel Computation, University of Vienna. [Available on the Web at multiple sites.][2]

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

Schrage, L. 1979, "A More Portable Fortran Random Number Generator," *ACM Transactions on Mathematical Software*, vol. 5, pp. 132–138.[4]

Park, S.K., and Miller, K.W. 1988, "Random Number Generators: Good Ones Are Hard to Find," *Communications of the ACM*, vol. 31, pp. 1192–1201.[5]

L'Ecuyer, P. 1997 "Uniform Random Number Generators: A Review," *Proceedings of the 1997 Winter Simulation Conference*, Andradóttir, S. et al., eds. (Piscataway, NJ: IEEE).[6]

Marsaglia, G. 1999, "Random Numbers for C: End, at Last?", posted 1999 January 20 to sci.stat.math.[7]

Marsaglia, G. 2003, "Diehard Battery of Tests of Randomness v0.2 beta," 2007+ at `http://www.cs.hku.hk/~diehard/`.[8]

Rukhin, A. et al. 2001, "A Statistical Test Suite for Random and Pseudorandom Number Generators", NIST Special Publication 800-22 (revised to May 15, 2001).[9]

Marsaglia, G. 2003, "Xorshift RNGs", *Journal of Statistical Software*, vol. 8, no. 14, pp. 1-6.[10]

Brent, R.P. 2004, "Note on Marsaglia's Xorshift Random Number Generators", *Journal of Statistical Software*, vol. 11, no. 5, pp. 1-5.[11]

L'Ecuyer, P. 1996, "Maximally Equidistributed Combined Tausworthe Generators," *Mathematics of Computation*, vol. 65, pp. 203-213.[12]

L'Ecuyer, P. 1999, "Tables of Maximally Equidistributed Combined LSFR Generators," *Mathematics of Computation*, vol. 68, pp. 261-269.[13]

Couture, R. and L'Ecuyer, P. 1997, "Distribution Properties of Multiply-with-Carry Random Number Generators," *Mathematics of Computation*, vol. 66, pp. 591-607.[14]

L'Ecuyer, P. 1999, "Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure", *Mathematics of Computation*, vol. 68, pp. 249-260.[15]

7.2 Completely Hashing a Large Array

We introduced the idea of a random hash or *hash function* in §7.1.4. Once in a while we might want a hash function that operates not on a single word, but on an entire array of length M. Being perfectionists, we want every single bit in the hashed output array to depend on every single bit in the given input array. One way to achieve this is to borrow structural concepts from algorithms as unrelated as the Data Encryption Standard (DES) and the Fast Fourier Transform (FFT)!

DES, like its progenitor cryptographic system LUCIFER, is a so-called "block product cipher" [1]. It acts on 64 bits of input by iteratively applying (16 times, in fact) a kind of highly nonlinear bit-mixing function. Figure 7.2.1 shows the flow of information in DES during this mixing. The function g, which takes 32 bits into 32 bits, is called the "cipher function." Meyer and Matyas [1] discuss the importance of the cipher function being nonlinear, as well as other design criteria.

DES constructs its cipher function g from an intricate set of bit permutations and table lookups acting on short sequences of consecutive bits. For our purposes, a different function g that can be rapidly computed in a high-level computer language is preferable. Such a function probably weakens the algorithm cryptographically. Our purposes are not, however, cryptographic: We want to find the fastest g, and the smallest number of iterations of the mixing procedure in Figure 7.2.1, such that our output random sequence passes the tests that are customarily applied to random number generators. The resulting algorithm is not DES, but rather a kind of "pseudo-DES," better suited to the purpose at hand.

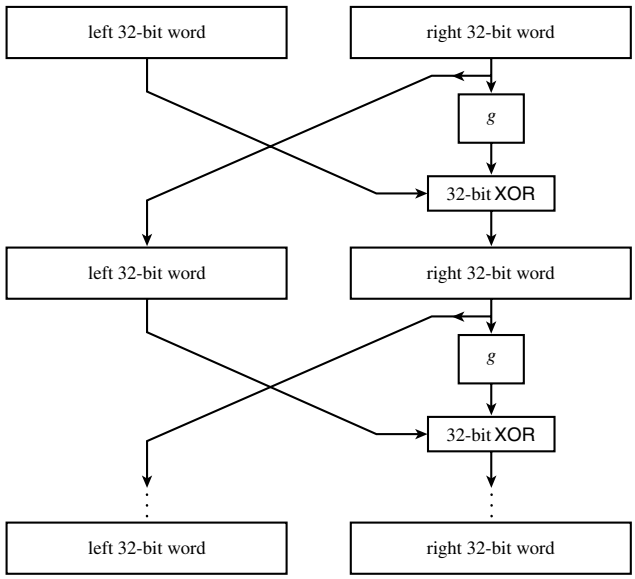

Figure 7.2.1. The Data Encryption Standard (DES) iterates a nonlinear function g on two 32-bit words, in the manner shown here (after Meyer and Matyas [1]).

Following the criterion mentioned above, that g should be nonlinear, we must give the integer multiply operation a prominent place in g. Confining ourselves to multiplying 16-bit operands into a 32-bit result, the general idea of g is to calculate the three distinct 32-bit products of the high and low 16-bit input half-words, and then to combine these, and perhaps additional fixed constants, by fast operations (e.g., add or exclusive-or) into a single 32-bit result.

There are only a limited number of ways of effecting this general scheme, allowing systematic exploration of the alternatives. Experimentation and tests of the randomness of the output lead to the sequence of operations shown in Figure 7.2.2. The few new elements in the figure need explanation: The values C_1 and C_2 are fixed constants, chosen randomly with the constraint that they have exactly 16 1-bits and 16 0-bits; combining these constants via exclusive-or ensures that the overall g has no bias toward 0- or 1-bits. The "reverse half-words" operation in Figure 7.2.2 turns out to be essential; otherwise, the very lowest and very highest bits are not properly mixed by the three multiplications.

It remains to specify the smallest number of iterations N_{it} that we can get away with. For purposes of this section, we recommend $N_{it} = 2$. We have not found any statistical deviation from randomness in sequences of up to 10^9 random deviates derived from this scheme. However, we include C_1 and C_2 constants for $N_{it} \leq 4$.

```
void psdes(Uint &lword, Uint &rword) {                                    hashall.h
```
Pseudo-DES hashing of the 64-bit word `(lword,rword)`. Both 32-bit arguments are returned hashed on all bits.
```
    const int NITER=2;
    static const Uint c1[4]={
        0xbaa96887L, 0x1e17d32cL, 0x03bcdc3cL, 0x0f33d1b2L};
    static const Uint c2[4]={
        0x4b0f3b58L, 0xe874f0c3L, 0x6955c5a6L, 0x55a7ca46L};
    Uint i,ia,ib,iswap,itmph=0,itmpl=0;
    for (i=0;i<NITER;i++) {
```
Perform `niter` iterations of DES logic, using a simpler (noncryptographic) nonlinear function instead of DES's.
```
        ia = (iswap=rword) ^ c1[i];
        itmpl = ia & 0xffff;
        itmph = ia >> 16;
```
The bit-rich constants `c1` and (below) `c2` guarantee lots of nonlinear mixing.

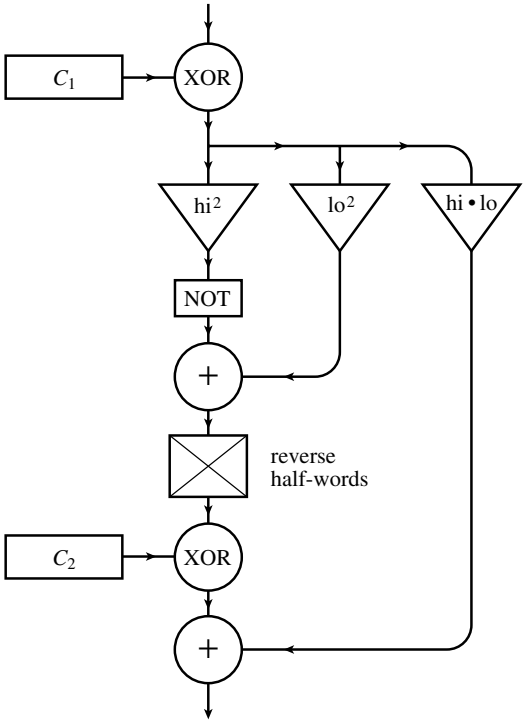

Figure 7.2.2. The nonlinear function g used by the routine `psdes`.

```
        ib=itmpl*itmpl+ ~(itmph*itmph);
        rword = lword ^ (((ia = (ib >> 16) |
            ((ib & 0xffff) << 16)) ^ c2[i])+itmpl*itmph);
        lword = iswap;
    }
}
```

Thus far, this doesn't seem to have much to do with completely hashing a large array. However, `psdes` gives us a building block, a routine for mutually hashing two arbitrary 32-bit integers. We now turn to the FFT concept of the *butterfly* to extend the hash to a whole array.

The butterfly is a particular algorithmic construct that applies to an array of length N, a power of 2. It brings every element into mutual communication with every other element in about $N \log_2 N$ operations. A useful metaphor is to imagine that one array element has a disease that infects any other element with which it has contact. Then the butterfly has two properties of interest here: (i) After its $\log_2 N$ stages, everyone has the disease. Furthermore, (ii) after j stages, 2^j elements are infected; there is never an "eye of the needle" or "necking down" of the communication path.

The butterfly is very simple to describe: In the first stage, every element in the first half of the array mutually communicates with its corresponding element in the second half of the array. Now recursively do this same thing to each of the halves, and so on. We can see by induction that every element now has a communication path to every other one: Obviously it works when $N = 2$. And if it works for N, it must also work for $2N$, because the first step gives every element a communication path into both its own and the other half of the array, after which it has, by assumption, a path everywhere.

We need to modify the butterfly slightly, so that our array size M does not have to be a power of 2. Let N be the next larger power of 2. We do the butterfly on the (virtual) size N, ignoring any communication with nonexistent elements larger than M. This, by itself, doesn't do the job, because the later elements in the first $N/2$ were not able to "infect" the second $N/2$ (and similarly at later recursive levels). However, if we do one extra communication between

elements of the first $N/2$ and second $N/2$ at the very end, then all missing communication paths are restored by traveling through the first $N/2$ elements.

The third line in the following code is an idiom that sets n to the next larger power of 2 greater or equal to m, a miniature masterpiece due to S.E. Anderson [2]. If you look closely, you'll see that it is itself a sort of butterfly, but now on bits!

hashall.h

```
void hashall(VecUint &arr) {
Replace the array arr by a same-sized hash, all of whose bits depend on all of the bits in arr.
Uses psdes for the mutual hash of two 32-bit words.
    Int m=arr.size(), n=m-1;
    n|=n>>1; n|=n>>2; n|=n>>4; n|=n>>8; n|=n>>16; n++;
    Incredibly, n is now the next power of 2 ≥ m.
    Int nb=n,nb2=n>>1,j,jb;
    if (n<2) throw("size must be > 1");
    while (nb > 1) {
        for (jb=0;jb<n-nb+1;jb+=nb)
            for (j=0;j<nb2;j++)
                if (jb+j+nb2 < m) psdes(arr[jb+j],arr[jb+j+nb2]);
        nb = nb2;
        nb2 >>= 1;
    }
    nb2 = n>>1;
    if (m != n) for (j=nb2;j<m;j++) psdes(arr[j],arr[j-nb2]);
    Final mix needed only if m is not a power of 2.
}
```

CITED REFERENCES AND FURTHER READING:

Meyer, C.H. and Matyas, S.M. 1982, *Cryptography: A New Dimension in Computer Data Security* (New York: Wiley).[1]

Zonst, A.E. 2000, *Understanding the FFT*, 2nd revised ed. (Titusville, FL: Citrus Press).

Anderson, S.E. 2005, "Bit Twiddling Hacks," 2007+ at `http://graphics.stanford.edu/~seander/bithacks.html` .[2]

Data Encryption Standard, 1977 January 15, Federal Information Processing Standards Publication, number 46 (Washington: U.S. Department of Commerce, National Bureau of Standards).

Guidelines for Implementing and Using the NBS Data Encryption Standard, 1981 April 1, Federal Information Processing Standards Publication, number 74 (Washington: U.S. Department of Commerce, National Bureau of Standards).

7.3 Deviates from Other Distributions

In §7.1 we learned how to generate random deviates with a uniform probability between 0 and 1, denoted U(0, 1). The probability of generating a number between x and $x + dx$ is

$$p(x)dx = \begin{cases} dx & 0 \le x < 1 \\ 0 & \text{otherwise} \end{cases} \tag{7.3.1}$$

and we write

$$x \sim \mathrm{U}(0,1) \tag{7.3.2}$$

As in §6.14, the symbol $\sim$ can be read as "is drawn from the distribution."

In this section, we learn how to generate random deviates drawn from other probability distributions, including all of those discussed in §6.14. Discussion of specific distributions is interleaved with the discussion of the general methods used.

7.3.1 Exponential Deviates

Suppose that we generate a uniform deviate x and then take some prescribed function of it, $y(x)$. The probability distribution of y, denoted $p(y)dy$, is determined by the fundamental transformation law of probabilities, which is simply

$$|p(y)dy| = |p(x)dx| \tag{7.3.3}$$

or

$$p(y) = p(x)\left|\frac{dx}{dy}\right| \tag{7.3.4}$$

As an example, take

$$y(x) = -\ln(x) \tag{7.3.5}$$

with $x \sim \mathrm{U}(0,1)$. Then

$$p(y)dy = \left|\frac{dx}{dy}\right| dy = e^{-y}dy \tag{7.3.6}$$

which is the exponential distribution with unit mean, Exponential (1), discussed in §6.14.5. This distribution occurs frequently in real life, usually as the distribution of waiting times between independent Poisson-random events, for example the radioactive decay of nuclei. You can also easily see (from 7.3.6) that the quantity y/β has the probability distribution $\beta e^{-\beta y}$, so

$$y/\beta \sim \text{Exponential}\,(\beta) \tag{7.3.7}$$

We can thus generate exponential deviates at a cost of about one uniform deviate, plus a logarithm, per call.

deviates.h

```
struct Expondev : Ran {
```
Structure for exponential deviates.
```
    Doub beta;
    Expondev(Doub bbeta, Ullong i) : Ran(i), beta(bbeta) {}
```
Constructor arguments are β and a random sequence seed.
```
    Doub dev() {
```
Return an exponential deviate.
```
        Doub u;
        do u = doub(); while (u == 0.);
        return -log(u)/beta;
    }
};
```

Our convention here and in the rest of this section is to derive the class for each kind of deviate from the uniform generator class `Ran`. We use the constructor to set the distribution's parameters and set the initial seed for the generator. We then provide a method `dev()` that returns a random deviate from the distribution.

7.3.2 Transformation Method in General

Let's see what is involved in using the above *transformation method* to generate some arbitrary desired distribution of y's, say one with $p(y) = f(y)$ for some positive function f whose integral is 1. According to (7.3.4), we need to solve the differential equation

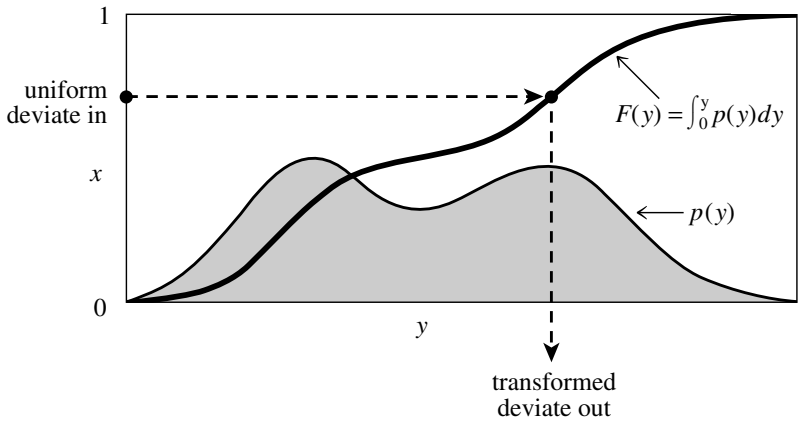

Figure 7.3.1. Transformation method for generating a random deviate y from a known probability distribution $p(y)$. The indefinite integral of $p(y)$ must be known and invertible. A uniform deviate x is chosen between 0 and 1. Its corresponding y on the definite-integral curve is the desired deviate.

$$\frac{dx}{dy} = f(y) \tag{7.3.8}$$

But the solution of this is just $x = F(y)$, where $F(y)$ is the indefinite integral of $f(y)$. The desired transformation that takes a uniform deviate into one distributed as $f(y)$ is therefore

$$y(x) = F^{-1}(x) \tag{7.3.9}$$

where F^{-1} is the inverse function to F. Whether (7.3.9) is feasible to implement depends on whether the *inverse function of the integral of f(y)* is itself feasible to compute, either analytically or numerically. Sometimes it is, and sometimes it isn't.

Incidentally, (7.3.9) has an immediate geometric interpretation: Since $F(y)$ is the area under the probability curve to the left of y, (7.3.9) is just the prescription: Choose a uniform random x, then find the value y that has that fraction x of probability area to its left, and return the value y. (See Figure 7.3.1.)

7.3.3 Logistic Deviates

Deviates from the logistic distribution, as discussed in §6.14.4, are readily generated by the transformation method, using equation (6.14.15). The cost is again dominated by one uniform deviate, and a logarithm, per logistic deviate.

deviates.h

```
struct Logisticdev : Ran {
Structure for logistic deviates.
    Doub mu,sig;
    Logisticdev(Doub mmu, Doub ssig, Ullong i) : Ran(i), mu(mmu), sig(ssig) {}
    Constructor arguments are μ, σ, and a random sequence seed.
    Doub dev() {
    Return a logistic deviate.
        Doub u;
        do u = doub(); while (u*(1.-u) == 0.);
        return mu + 0.5513288954217920 50*sig*log(u/(1.-u));
    }
};
```

7.3.4 Normal Deviates by Transformation (Box-Muller)

Transformation methods generalize to more than one dimension. If $x_1, x_2, \ldots$ are random deviates with a *joint* probability distribution $p(x_1, x_2, \ldots)dx_1 dx_2 \ldots$, and if $y_1, y_2, \ldots$ are each functions of all the x's (same number of y's as x's), then the joint probability distribution of the y's is

$$p(y_1, y_2, \ldots)dy_1 dy_2 \ldots = p(x_1, x_2, \ldots)\left|\frac{\partial(x_1, x_2, \ldots)}{\partial(y_1, y_2, \ldots)}\right| dy_1 dy_2 \ldots \tag{7.3.10}$$

where $|\partial(\ \ )/\partial(\ \ )|$ is the Jacobian determinant of the x's with respect to the y's (or the reciprocal of the Jacobian determinant of the y's with respect to the x's).

An important historical example of the use of (7.3.10) is the *Box-Muller* method for generating random deviates with a normal (Gaussian) distribution (§6.14.1):

$$p(y)dy = \frac{1}{\sqrt{2\pi}} e^{-y^2/2} dy \tag{7.3.11}$$

Consider the transformation between two uniform deviates on (0,1), x_1, x_2, and two quantities y_1, y_2,

$$\begin{aligned} y_1 &= \sqrt{-2\ln x_1}\cos 2\pi x_2 \\ y_2 &= \sqrt{-2\ln x_1}\sin 2\pi x_2 \end{aligned} \tag{7.3.12}$$

Equivalently we can write

$$\begin{aligned} x_1 &= \exp\left[-\frac{1}{2}(y_1^2 + y_2^2)\right] \\ x_2 &= \frac{1}{2\pi}\arctan\frac{y_2}{y_1} \end{aligned} \tag{7.3.13}$$

Now the Jacobian determinant can readily be calculated (try it!):

$$\frac{\partial(x_1, x_2)}{\partial(y_1, y_2)} = \begin{vmatrix} \frac{\partial x_1}{\partial y_1} & \frac{\partial x_1}{\partial y_2} \\ \frac{\partial x_2}{\partial y_1} & \frac{\partial x_2}{\partial y_2} \end{vmatrix} = -\left[\frac{1}{\sqrt{2\pi}} e^{-y_1^2/2}\right]\left[\frac{1}{\sqrt{2\pi}} e^{-y_2^2/2}\right] \tag{7.3.14}$$

Since this is the product of a function of y_2 alone and a function of y_1 alone, we see that each y is independently distributed according to the normal distribution (7.3.11).

One further trick is useful in applying (7.3.12). Suppose that, instead of picking uniform deviates x_1 and x_2 in the unit square, we instead pick v_1 and v_2 as the ordinate and abscissa of a random point inside the unit circle around the origin. Then the sum of their squares, $R^2 \equiv v_1^2 + v_2^2$, is a uniform deviate, which can be used for x_1, while the angle that (v_1, v_2) defines with respect to the v_1-axis can serve as the random angle $2\pi x_2$. What's the advantage? It's that the cosine and sine in (7.3.12) can now be written as $v_1/\sqrt{R^2}$ and $v_2/\sqrt{R^2}$, obviating the trigonometric function calls! (In the next section we will generalize this trick considerably.)

Code for generating normal deviates by the Box-Muller method follows. Consider it for pedagogical use only, because a significantly faster method for generating normal deviates is coming, below, in §7.3.9.

deviates.h

```
struct Normaldev_BM : Ran {
```
Structure for normal deviates.
```
    Doub mu,sig;
    Doub storedval;
    Normaldev_BM(Doub mmu, Doub ssig, Ullong i)
    : Ran(i), mu(mmu), sig(ssig), storedval(0.) {}
```
Constructor arguments are μ, σ, and a random sequence seed.
```
    Doub dev() {
```
Return a normal deviate.
```
        Doub v1,v2,rsq,fac;
        if (storedval == 0.) {                  We don't have an extra deviate handy, so
            do {
                v1=2.0*doub()-1.0;              pick two uniform numbers in the square ex-
                v2=2.0*doub()-1.0;                  tending from -1 to +1 in each direction,
                rsq=v1*v1+v2*v2;                see if they are in the unit circle,
            } while (rsq >= 1.0 || rsq == 0.0);             or try again.
            fac=sqrt(-2.0*log(rsq)/rsq);        Now make the Box-Muller transformation to
            storedval = v1*fac;                     get two normal deviates. Return one and
            return mu + sig*v2*fac;                 save the other for next time.
        } else {                                We have an extra deviate handy,
            fac = storedval;
            storedval = 0.;
            return mu + sig*fac;                so return it.
        }
    }
};
```

7.3.5 Rayleigh Deviates

The *Rayleigh distribution* is defined for positive z by

$$p(z)dz = z \exp\left(-\tfrac{1}{2}z^2\right) dz \qquad (z > 0) \tag{7.3.15}$$

Since the indefinite integral can be done analytically, and the result easily inverted, a simple transformation method from a uniform deviate x results:

$$z = \sqrt{-2 \ln x}, \quad x \sim \mathrm{U}(0, 1) \tag{7.3.16}$$

A Rayleigh deviate z can also be generated from two normal deviates y_1 and y_2 by

$$z = \sqrt{y_1^2 + y_2^2}, \quad y_1, y_2 \sim \mathrm{N}(0, 1) \tag{7.3.17}$$

Indeed, the relation between equations (7.3.16) and (7.3.17) is immediately evident in the equation for the Box-Muller method, equation (7.3.12), if we square and sum that method's two normal deviates y_1 and y_2.

7.3.6 Rejection Method

The *rejection method* is a powerful, general technique for generating random deviates whose distribution function $p(x)dx$ (probability of a value occurring between x and $x + dx$) is known and computable. The rejection method does not require that the cumulative distribution function (indefinite integral of $p(x)$) be readily computable, much less the inverse of that function — which was required for the transformation method in the previous section.

The rejection method is based on a simple geometrical argument (Figure 7.3.2):

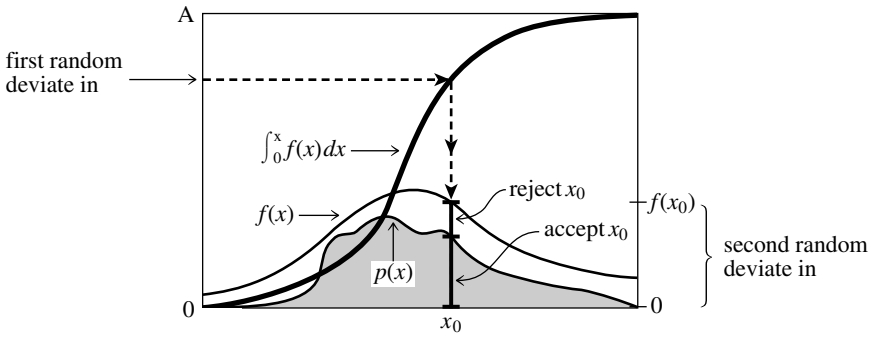

Figure 7.3.2. Rejection method for generating a random deviate x from a known probability distribution $p(x)$ that is everywhere less than some other function $f(x)$. The transformation method is first used to generate a random deviate x of the distribution f (compare Figure 7.3.1). A second uniform deviate is used to decide whether to accept or reject that x. If it is rejected, a new deviate of f is found, and so on. The ratio of accepted to rejected points is the ratio of the area under p to the area between p and f.

Draw a graph of the probability distribution $p(x)$ that you wish to generate, so that the area under the curve in any range of x corresponds to the desired probability of generating an x in that range. If we had some way of choosing a random point *in two dimensions*, with uniform probability in the *area* under your curve, then the x value of that random point would have the desired distribution.

Now, on the same graph, draw any other curve $f(x)$ that has finite (not infinite) area and lies everywhere *above* your original probability distribution. (This is always possible, because your original curve encloses only unit area, by definition of probability.) We will call this $f(x)$ the *comparison function*. Imagine now that you have some way of choosing a random point in two dimensions that is uniform in the area under the comparison function. Whenever that point lies outside the area under the original probability distribution, we will *reject* it and choose another random point. Whenever it lies inside the area under the original probability distribution, we will *accept* it.

It should be obvious that the accepted points are uniform in the accepted area, so that their x values have the desired distribution. It should also be obvious that the fraction of points rejected just depends on the ratio of the area of the comparison function to the area of the probability distribution function, not on the details of shape of either function. For example, a comparison function whose area is less than 2 will reject fewer than half the points, even if it approximates the probability function very badly at some values of x, e.g., remains finite in some region where $p(x)$ is zero.

It remains only to suggest how to choose a uniform random point in two dimensions under the comparison function $f(x)$. A variant of the transformation method (§7.3) does nicely: Be sure to have chosen a comparison function whose indefinite integral is known analytically, and is also analytically invertible to give x as a function of "area under the comparison function to the left of x." Now pick a uniform deviate between 0 and A, where A is the total area under $f(x)$, and use it to get a corresponding x. Then pick a uniform deviate between 0 and $f(x)$ as the y value for the two-dimensional point. Finally, accept or reject according to whether it is respectively less than or greater than $p(x)$.

So, to summarize, the rejection method for some given $p(x)$ requires that one find, once and for all, some reasonably good comparison function $f(x)$. Thereafter,

each deviate generated requires two uniform random deviates, one evaluation of f (to get the coordinate y) and one evaluation of p (to decide whether to accept or reject the point x, y). Figure 7.3.1 illustrates the whole process. Then, of course, this process may need to be repeated, on the average, A times before the final deviate is obtained.

7.3.7 Cauchy Deviates

The "further trick" described following equation (7.3.14) in the context of the Box-Muller method is now seen to be a rejection method for getting trigonometric functions of a uniformly random angle. If we combine this with the explicit formula, equation (6.14.6), for the inverse cdf of the Cauchy distribution (see §6.14.2), we can generate Cauchy deviates quite efficiently.

deviates.h

```
struct Cauchydev : Ran {
Structure for Cauchy deviates.
    Doub mu,sig;
    Cauchydev(Doub mmu, Doub ssig, Ullong i) : Ran(i), mu(mmu), sig(ssig) {}
    Constructor arguments are μ, σ, and a random sequence seed.
    Doub dev() {
    Return a Cauchy deviate.
        Doub v1,v2;
        do {                                   Find a random point in the unit semicircle.
            v1=2.0*doub()-1.0;
            v2=doub();
        } while (SQR(v1)+SQR(v2) >= 1. || v2 == 0.);
        return mu + sig*v1/v2;                 Ratio of its coordinates is the tangent of a
    }                                              random angle.
};
```

7.3.8 Ratio-of-Uniforms Method

In finding Cauchy deviates, we took the ratio of two uniform deviates chosen to lie within the unit circle. If we generalize to shapes other than the unit circle, and combine it with the principle of the rejection method, a powerful variant emerges. Kinderman and Monahan [1] showed that deviates of virtually *any* probability distribution $p(x)$ can be generated by the following rather amazing prescription:

- Construct the region in the (u, v) plane bounded by $0 \le u \le [p(v/u)]^{1/2}$.
- Choose two deviates, u and v, that lie uniformly in this region.
- Return v/u as the deviate.

Proof: We can represent the ordinary rejection method by the equation in the (x, p) plane,

$$p(x)dx = \int_{p'=0}^{p'=p(x)} dp'dx \tag{7.3.18}$$

Since the integrand is 1, we are justified in sampling uniformly in (x, p') as long as p' is within the limits of the integral (that is, $0 < p' < p(x)$). Now make the change of variable

$$\begin{aligned} \frac{v}{u} &= x \\ u^2 &= p \end{aligned} \tag{7.3.19}$$

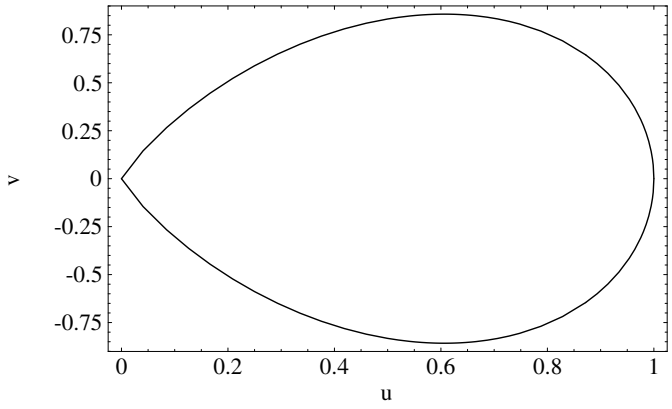

Figure 7.3.3. Ratio-of-uniforms method. The interior of this teardrop shape is the acceptance region for the normal distribution: If a random point is chosen inside this region, then the ratio v/u will be a normal deviate.

Then equation (7.3.18) becomes

$$p(x)dx = \int_{p'=0}^{p'=p(x)} dp'dx = \int_{u=0}^{u=\sqrt{p(x)}} \frac{\partial(p,x)}{\partial(u,v)} du\, dv = 2\int_{u=0}^{u=\sqrt{p(v/u)}} du\, dv \tag{7.3.20}$$

because (as you can work out) the Jacobian determinant is the constant 2. Since the new integrand is constant, uniform sampling in (u, v) with the limits indicated for u is equivalent to the rejection method in (x, p).

The above limits on u very often define a region that is "teardrop" shaped. To see why, note that the locii of constant $x = v/u$ are radial lines. Along each radial, the acceptance region goes from the origin to a point where $u^2 = p(x)$. Since most probability distributions go to zero for both large and small x, the acceptance region accordingly shrinks toward the origin along radials, producing a teardrop. Of course, it is the exact shape of this teardrop that matters. Figure 7.3.3 shows the shape of the acceptance region for the case of the normal distribution.

Typically this *ratio-of-uniforms* method is used when the desired region can be closely bounded by a rectangle, parallelogram, or some other shape that is easy to sample uniformly. Then, we go from sampling the easy shape to sampling the desired region by rejection of points outside the desired region.

An important adjunct to the ratio-of-uniforms method is the idea of a *squeeze*. A squeeze is any easy-to-compute shape that tightly bounds the region of acceptance of a rejection method, either from the inside or from the outside. Best of all is when you have squeezes on both sides. Then you can immediately reject points that are outside the outer squeeze and immediately accept points that are inside the inner squeeze. Only when you have the bad luck of drawing a point between the two squeezes do you actually have to do the more lengthy computation of comparing with the actual rejection boundary. Squeezes are useful both in the ordinary rejection method and in the ratio-of-uniforms method.

7.3.9 Normal Deviates by Ratio-of-Uniforms

Leva [2] has given an algorithm for normal deviates that uses the ratio-of-uniforms method with great success. He uses quadratic curves to provide both inner

and outer squeezes that hug the desired region in the (u, v) plane (Figure 7.3.3). Only about 1% of the time is it necessary to calculate an exact boundary (requiring a logarithm).

The resulting code looks so simple and "un-transcendental" that it may be hard to believe that exact normal deviates are generated. But they are!

deviates.h

```
struct Normaldev : Ran {
Structure for normal deviates.
    Doub mu,sig;
    Normaldev(Doub mmu, Doub ssig, Ullong i)
    : Ran(i), mu(mmu), sig(ssig){}
    Constructor arguments are μ, σ, and a random sequence seed.
    Doub dev() {
    Return a normal deviate.
        Doub u,v,x,y,q;
        do {
            u = doub();
            v = 1.7156*(doub()-0.5);
            x = u - 0.449871;
            y = abs(v) + 0.386595;
            q = SQR(x) + y*(0.19600*y-0.25472*x);
        } while (q > 0.27597
            && (q > 0.27846 || SQR(v) > -4.*log(u)*SQR(u)));
        return mu + sig*v/u;
    }
};
```

Note that the `while` clause makes use of C's (and C++'s) guarantee that logical expressions are evaluated conditionally: If the first operand is sufficient to determine the outcome, the second is not evaluated at all. With these rules, the logarithm is evaluated only when `q` is between 0.27597 and 0.27846.

On average, each normal deviate uses 2.74 uniform deviates. By the way, even though the various constants are given only to six digits, the method is exact (to full double precision). Small perturbations of the bounding curves are of no consequence. The accuracy is implicit in the (rare) evaluations of the exact boundary.

7.3.10 Gamma Deviates

The distribution Gamma(α, β) was described in §6.14.9. The β parameter enters only as a scaling,

$$\text{Gamma}(\alpha, \beta) \cong \frac{1}{\beta}\text{Gamma}(\alpha, 1) \tag{7.3.21}$$

(Translation: To generate a Gamma(α, β) deviate, generate a Gamma$(\alpha, 1)$ deviate and divide it by β.)

If α is a small positive integer, a fast way to generate $x \sim \text{Gamma}(\alpha, 1)$ is to use the fact that it is distributed as the waiting time to the αth event in a Poisson random process of unit mean. Since the time between two consecutive events is just the exponential distribution Exponential (1), you can simply add up α exponentially distributed waiting times, i.e., logarithms of uniform deviates. Even better, since the sum of logarithms is the logarithm of the product, you really only have to compute the product of a uniform deviates and then take the log. Because this is such a special case, however, we don't include it in the code below.

When $\alpha < 1$, the gamma distribution's density function is not bounded, which is inconvenient. However, it turns out [4] that if

$$y \sim \text{Gamma}(\alpha + 1, 1), \qquad u \sim \text{Uniform}(0, 1) \tag{7.3.22}$$

then

$$y u^{1/\alpha} \sim \text{Gamma}(\alpha, 1) \tag{7.3.23}$$

We will use this in the code below.

For $\alpha > 1$, Marsaglia and Tsang [5] give an elegant rejection method based on a simple transformation of the gamma distribution combined with a squeeze. After transformation, the gamma distribution can be bounded by a Gaussian curve whose area is never more than 5% greater than that of the gamma curve. The cost of a gamma deviate is thus only a little more than the cost of the normal deviate that is used to sample the comparison function. The following code gives the precise formulation; see the original paper for a full explanation.

deviates.h

```
struct Gammadev : Normaldev {
```
Structure for gamma deviates.
```
    Doub alph, oalph, bet;
    Doub a1,a2;
    Gammadev(Doub aalph, Doub bbet, Ullong i)
    : Normaldev(0.,1.,i), alph(aalph), oalph(aalph), bet(bbet) {
```
Constructor arguments are α, β, and a random sequence seed.
```
        if (alph <= 0.) throw("bad alph in Gammadev");
        if (alph < 1.) alph += 1.;
        a1 = alph-1./3.;
        a2 = 1./sqrt(9.*a1);
    }
    Doub dev() {
```
Return a gamma deviate by the method of Marsaglia and Tsang.
```
        Doub u,v,x;
        do {
            do {
                x = Normaldev::dev();
                v = 1. + a2*x;
            } while (v <= 0.);
            v = v*v*v;
            u = doub();
        } while (u > 1. - 0.331*SQR(SQR(x)) &&
            log(u) > 0.5*SQR(x) + a1*(1.-v+log(v)));
```
Rarely evaluated.
```
        if (alph == oalph) return a1*v/bet;
        else {
```
Case where $\alpha < 1$, per Ripley.
```
            do u=doub(); while (u == 0.);
            return pow(u,1./oalph)*a1*v/bet;
        }
    }
};
```

There exists a sum rule for gamma deviates. If we have a set of independent deviates y_i with possibly different α_i's, but sharing a common value of β,

$$y_i \sim \text{Gamma}(\alpha_i, \beta) \tag{7.3.24}$$

then their sum is also a gamma deviate,

$$y \equiv \sum_i y_i \sim \text{Gamma}(\alpha_T, \beta), \qquad \alpha_T = \sum_i \alpha_i \tag{7.3.25}$$

If the α_i's are integers, you can see how this relates to the discussion of Poisson waiting times above.

7.3.11 Distributions Easily Generated by Other Deviates

From normal, gamma and uniform deviates, we get a bunch of other distributions for free. Important: When you are going to combine their results, be sure that all distinct instances of `Normaldist`, `Gammadist`, and `Ran` have different random seeds! (`Ran` and its derived classes are sufficiently robust that seeds $i, i+1, \ldots$ are fine.)

Chi-Square Deviates (cf. §6.14.8)

This one is easy:

$$\text{Chisquare}(\nu) \cong \text{Gamma}\left(\frac{\nu}{2}, \frac{1}{2}\right) \cong 2\,\text{Gamma}\left(\frac{\nu}{2}, 1\right) \tag{7.3.26}$$

Student-t Deviates (cf. §6.14.3)

Deviates from the Student-t distribution can be generated by a method very similar to the Box-Muller method. The analog of equation (7.3.12) is

$$y = \sqrt{\nu(u_1^{-2/\nu} - 1)}\cos 2\pi u_2 \tag{7.3.27}$$

If u_1 and u_2 are independently uniform, U(0, 1), then

$$y \sim \text{Student}(\nu, 0, 1) \tag{7.3.28}$$

or

$$\mu + \sigma y \sim \text{Student}(\nu, \mu, \sigma) \tag{7.3.29}$$

Unfortunately, you can't do the Box-Muller trick of getting two deviates at a time, because the Jacobian determinant analogous to equation (7.3.14) does not factorize. You might want to use the polar method anyway, just to get $\cos 2\pi u_2$, but its advantage is now not so large.

An alternative method uses the quotients of normal and gamma deviates. If we have

$$x \sim \text{N}(0, 1), \qquad y \sim \text{Gamma}\left(\frac{\nu}{2}, \frac{1}{2}\right) \tag{7.3.30}$$

then

$$x\sqrt{\nu/y} \sim \text{Student}(\nu, 0, 1) \tag{7.3.31}$$

Beta Deviates (cf. §6.14.11)

If

$$x \sim \text{Gamma}(\alpha, 1), \qquad y \sim \text{Gamma}(\beta, 1) \tag{7.3.32}$$

then

$$\frac{x}{x+y} \sim \text{Beta}(\alpha, \beta) \tag{7.3.33}$$

F-Distribution Deviates (cf. §6.14.10)

If

$$x \sim \text{Beta}(\tfrac{1}{2}\nu_1, \tfrac{1}{2}\nu_2) \tag{7.3.34}$$

(see equation 7.3.33), then

$$\frac{\nu_2 x}{\nu_1(1-x)} \sim \text{F}(\nu_1, \nu_2) \tag{7.3.35}$$

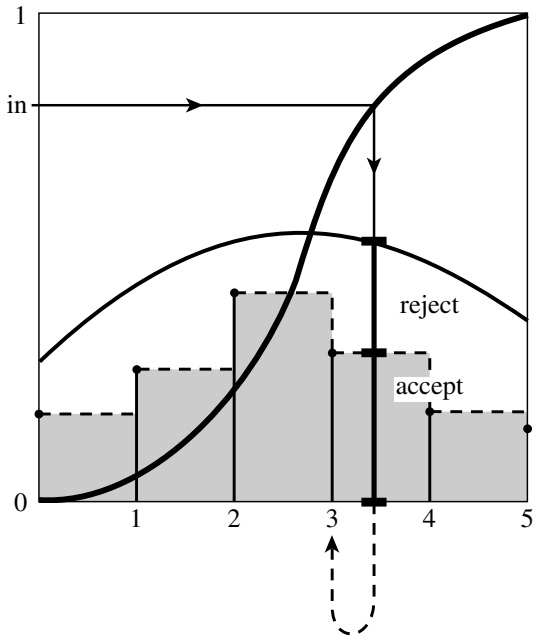

Figure 7.3.4. Rejection method as applied to an integer-valued distribution. The method is performed on the step function shown as a dashed line, yielding a real-valued deviate. This deviate is rounded down to the next lower integer, which is output.

7.3.12 Poisson Deviates

The Poisson distribution, Poisson(λ), previously discussed in §6.14.13, is a discrete distribution, so its deviates will be integers, k. To use the methods already discussed, it is convenient to convert the Poisson distribution into a continuous distribution by the following trick: Consider the finite probability $p(k)$ as being spread out uniformly into the interval from k to $k+1$. This defines a continuous distribution $q_\lambda(k)dk$ given by

$$q_\lambda(k)dk = \frac{\lambda^{\lfloor k \rfloor} e^{-\lambda}}{\lfloor k \rfloor !} dk \tag{7.3.36}$$

where $\lfloor k \rfloor$ represents the largest integer $\le k$. If we now use a rejection method, or any other method, to generate a (noninteger) deviate from (7.3.36), and then take the integer part of that deviate, it will be as if drawn from the discrete Poisson distribution. (See Figure 7.3.4.) This trick is general for any integer-valued probability distribution. Instead of the "floor" operator, one can equally well use "ceiling" or "nearest" — anything that spreads the probability over a unit interval.

For λ large enough, the distribution (7.3.36) is qualitatively bell-shaped (albeit with a bell made out of small, square steps). In that case, the ratio-of-uniforms method works well. It is not hard to find simple inner and outer squeezes in the (u, v) plane of the form $v^2 = Q(u)$, where $Q(u)$ is a simple polynomial in u. The only trick is to allow a big enough gap between the squeezes to enclose the true, jagged, boundaries for all values of λ. (Look ahead to Figure 7.3.5 for a similar example.)

For intermediate values of λ, the jaggedness is so large as to render squeezes impractical, but the ratio-of-uniforms method, unadorned, still works pretty well.

For small λ, we can use an idea similar to that mentioned above for the gamma distribution in the case of integer a. When the sum of independent exponential

deviates first exceeds λ, their number (less 1) is a Poisson deviate k. Also, as explained for the gamma distribution, we can multiply uniform deviates from U(0, 1) instead of adding deviates from Exponential (1).

These ideas produce the following routine.

deviates.h

```
struct Poissondev : Ran {
```
Structure for Poisson deviates.
```
    Doub lambda, sqlam, loglam, lamexp, lambold;
    VecDoub logfact;
    Int swch;
    Poissondev(Doub llambda, Ullong i) : Ran(i), lambda(llambda),
        logfact(1024,-1.), lambold(-1.) {}
```
Constructor arguments are λ and a random sequence seed.
```
    Int dev() {
```
Return a Poisson deviate using the most recently set value of λ.
```
        Doub u,u2,v,v2,p,t,lfac;
        Int k;
        if (lambda < 5.) {                          Will use product of uniforms method.
            if (lambda != lambold) lamexp=exp(-lambda);
            k = -1;
            t=1.;
            do {
                ++k;
                t *= doub();
            } while (t > lamexp);
        } else {                                    Will use ratio-of-uniforms method.
            if (lambda != lambold) {
                sqlam = sqrt(lambda);
                loglam = log(lambda);
            }
            for (;;) {
                u = 0.64*doub();
                v = -0.68 + 1.28*doub();
                if (lambda > 13.5) {                Outer squeeze for fast rejection.
                    v2 = SQR(v);
                    if (v >= 0.) {if (v2 > 6.5*u*(0.64-u)*(u+0.2)) continue;}
                    else {if (v2 > 9.6*u*(0.66-u)*(u+0.07)) continue;}
                }
                k = Int(floor(sqlam*(v/u)+lambda+0.5));
                if (k < 0) continue;
                u2 = SQR(u);
                if (lambda > 13.5) {                Inner squeeze for fast acceptance.
                    if (v >= 0.) {if (v2 < 15.2*u2*(0.61-u)*(0.8-u)) break;}
                    else {if (v2 < 6.76*u2*(0.62-u)*(1.4-u)) break;}
                }
                if (k < 1024) {
                    if (logfact[k] < 0.) logfact[k] = gammln(k+1.);
                    lfac = logfact[k];
                } else lfac = gammln(k+1.);
                p = sqlam*exp(-lambda + k*loglam - lfac);    Only when we must.
                if (u2 < p) break;
            }
        }
        lambold = lambda;
        return k;
    }
    Int dev(Doub llambda) {
```
Reset λ and then return a Poisson deviate.
```
        lambda = llambda;
        return dev();
    }
};
```

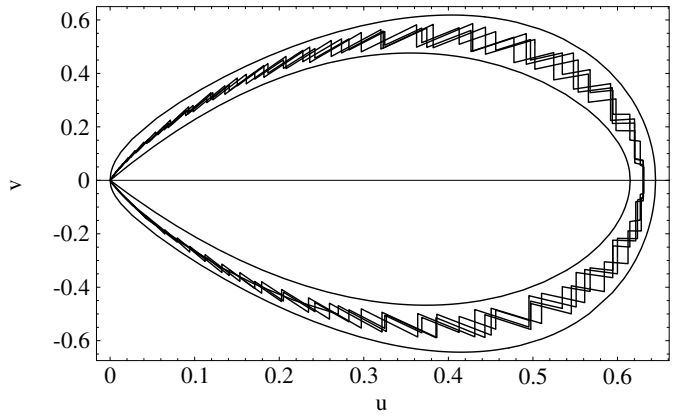

Figure 7.3.5. Ratio-of-uniforms method as applied to the generation of binomial deviates. Points are chosen randomly in the (u, v)-plane. The smooth curves are inner and outer squeezes. The jagged curves correspond to various binomial distributions with $n > 64$ and $np > 30$. An evaluation of the binomial probability is required only when the random point falls between the smooth curves.

In the regime $\lambda > 13.5$, the above code uses about 3.3 uniform deviates per output Poisson deviate and does 0.4 evaluations of the exact probability (costing an exponential and, for large k, a call to `gammln`).

`Poissondev` is slightly faster if you draw many deviates with the same value λ, using the `dev` function with no arguments, than if you vary λ on each call, using the one-argument overloaded form of `dev` (which is provided for just that purpose). The difference is just an extra exponential ($\lambda < 5$) or square root and logarithm ($\lambda \geq 5$). Note also the object's table of previously computed log-factorials. If your λ's are as large as $\sim 10^3$, you might want to make the table larger.

7.3.13 Binomial Deviates

The generation of binomial deviates $k \sim \text{Binomial}(n, p)$ involves many of the same ideas as for Poisson deviates. The distribution is again integer-valued, so we use the same trick to convert it into a stepped continuous distribution. We can always restrict attention to the case $p \leq 0.5$, since the distribution's symmetries let us trivially recover the case $p > 0.5$.

When $n > 64$ and $np > 30$, we use the ratio-of-uniforms method, with squeezes shown in Figure 7.3.5. The cost is about 3.2 uniform deviates, plus 0.4 evaluations of the exact probability, per binomial deviate.

It would be foolish to waste much thought on the case where $n > 64$ and $np < 30$, because it is so easy simply to tabulate the cdf, say for $0 \leq k < 64$, and then loop over k's until the right one is found. (A bisection search, implemented below, is even better.) With a cdf table of length 64, the neglected probability at the end of the table is never larger than $\sim 10^{-20}$. (At 10^9 deviates per second, you could run 3000 years before losing a deviate.)

What is left is the interesting case $n < 64$, which we will explore in some detail, because it demonstrates the important concept of *bit-parallel random comparison*.

Analogous to the methods for gamma deviates with small integer a and for Poisson deviates with small λ, is this direct method for binomial deviates: Generate n uniform deviates in $\text{U}(0, 1)$. Count the number of them $< p$. Return the count as

$k \sim \text{Binomial}(n, p)$. Indeed this is essentially the definition of a binomial process!

The problem with the direct method is that it seems to require n uniform deviates, even when the mean value of k is much smaller. Would you be surprised if we told you that for $n \leq 64$ you can achieve the same goal with at most *seven* 64-bit uniform deviates, on average? Here is how.

Expand $p < 1$ into its first 5 bits, plus a residual,

$$p = b_1 2^{-1} + b_2 2^{-2} + \cdots + b_5 2^{-5} + p_r 2^{-5} \tag{7.3.37}$$

where each b_i is 0 or 1, and $0 \leq p_r \leq 1$.

Now imagine that you have generated and stored 64 uniform U(0, 1) deviates, and that the 64-bit word P displays just the first bit of each of the 64. Compare each bit of P to b_1. If the bits are the same, then we don't yet know whether that uniform deviate is less than or greater than p. But if the bits are *different*, then we know that the generator is less than p (in the case that $b_1 = 1$) or greater than p (in the case that $b_1 = 0$). If we keep a mask of "known" versus "unknown" cases, we can do these comparisons in a bit-parallel manner by bitwise logical operations (see code below to learn how). Now move on to the second bit, b_2, in the same way. At each stage we change half the remaining unknowns to knowns. After five stages (for $n = 64$) there will be two remaining unknowns, on average, each of which we finish off by generating a new uniform and comparing it to p_r. (This requires a loop through the 64 bits; but since C++ has no bitwise "popcount" operation, we are stuck doing such a loop anyway. If you can do popcounts, you may be better off just doing more stages until the unknowns mask is zero.)

The trick is that the bits used in the five stages are not actually the leading five bits of 64 generators, they are just five independent 64-bit random integers. The number five was chosen because it minimizes $64 \times 2^{-j} + j$, the expected number of deviates needed.

So, the code for binomial deviates ends up with three separate methods: bit-parallel direct, cdf lookup (by bisection), and squeezed ratio-of-uniforms.

deviates.h

```
struct Binomialdev : Ran {
Structure for binomial deviates.
    Doub pp,p,pb,expnp,np,glnp,plog,pclog,sq;
    Int n,swch;
    Ullong uz,uo,unfin,diff,rltp;
    Int pbits[5];
    Doub cdf[64];
    Doub logfact[1024];
    Binomialdev(Int nn, Doub ppp, Ullong i) : Ran(i), pp(ppp), n(nn) {
    Constructor arguments are n, p, and a random sequence seed.
        Int j;
        pb = p = (pp <= 0.5 ? pp : 1.0-pp);
        if (n <= 64) {                          Will use bit-parallel direct method.
            uz=0;
            uo=0xffffffffffffffffLL;
            rltp = 0;
            for (j=0;j<5;j++) pbits[j] = 1 & ((Int)(pb *= 2.));
            pb -= floor(pb);                    Leading bits of p (above) and remaining
            swch = 0;                               fraction.
        } else if (n*p < 30.) {                 Will use precomputed cdf table.
            cdf[0] = exp(n*log(1-p));
            for (j=1;j<64;j++) cdf[j] =  cdf[j-1] + exp(gammln(n+1.)
                -gammln(j+1.)-gammln(n-j+1.)+j*log(p)+(n-j)*log(1.-p));
            swch = 1;
```

```
        } else {                                Will use ratio-of-uniforms method.
            np = n*p;
            glnp=gammln(n+1.);
            plog=log(p);
            pclog=log(1.-p);
            sq=sqrt(np*(1.-p));
            if (n < 1024) for (j=0;j<=n;j++) logfact[j] = gammln(j+1.);
            swch = 2;
        }
    }
    Int dev() {
    Return a binomial deviate.
        Int j,k,kl,km;
        Doub y,u,v,u2,v2,b;
        if (swch == 0) {
            unfin = uo;                         Mark all bits as "unfinished."
            for (j=0;j<5;j++) {                 Compare with first five bits of p.
                diff = unfin & (int64()^(pbits[j]? uo : uz));   Mask of diff.
                if (pbits[j]) rltp |= diff;     Set bits to 1, meaning ran < p.
                else rltp = rltp & ~diff;       Set bits to 0, meaning ran > p.
                unfin = unfin & ~diff;          Update unfinished status.
            }
            k=0;                                Now we just count the events.
            for (j=0;j<n;j++) {
                if (unfin & 1) {if (doub() < pb) ++k;}      Clean up unresolved cases,
                else {if (rltp & 1) ++k;}                   or use bit answer.
                unfin >>= 1;
                rltp >>= 1;
            }
        } else if (swch == 1) {                 Use stored cdf.
            y = doub();
            kl = -1;
            k = 64;
            while (k-kl>1) {
                km = (kl+k)/2;
                if (y < cdf[km]) k = km;
                else kl = km;
            }
        } else {                                Use ratio-of-uniforms method.
            for (;;) {
                u = 0.645*doub();
                v = -0.63 + 1.25*doub();
                v2 = SQR(v);
                Try squeeze for fast rejection:
                if (v >= 0.) {if (v2 > 6.5*u*(0.645-u)*(u+0.2)) continue;}
                else {if (v2 > 8.4*u*(0.645-u)*(u+0.1)) continue;}
                k = Int(floor(sq*(v/u)+np+0.5));
                if (k < 0) continue;
                u2 = SQR(u);
                Try squeeze for fast acceptance:
                if (v >= 0.) {if (v2 < 12.25*u2*(0.615-u)*(0.92-u)) break;}
                else {if (v2 < 7.84*u2*(0.615-u)*(1.2-u)) break;}
                b = sq*exp(glnp+k*plog+(n-k)*pclog   Only when we must.
                    - (n < 1024 ? logfact[k]+logfact[n-k]
                        : gammln(k+1.)+gammln(n-k+1.)));
                if (u2 < b) break;
            }
        }
        if (p != pp) k = n - k;
        return k;
    }
};
```

If you are in a situation where you are drawing only one or a few deviates each for many different values of n and/or p, you'll need to restructure the code so that n and p can be changed without creating a new instance of the object and without reinitializing the underlying `Ran` generator.

7.3.14 When You Need Greater Speed

In particular situations you can cut some corners to gain greater speed. Here are some suggestions.

- All of the algorithms in this section can be speeded up significantly by using `Ranq1` in §7.1 instead of `Ran`. We know of no reason not to do this. You can gain some further speed by coding `Ranq1`'s algorithm inline, thus eliminating the function calls.
- If you are using `Poissondev` or `Binomialdev` with large values of λ or n, then the above codes revert to calling `gammln`, which is slow. You can instead increase the length of the stored tables.
- For Poisson deviates with $\lambda < 20$, you may want to use a stored table of cdfs combined with bisection to find the value of k. The code in `Binomialdev` shows how to do this.
- If your need is for binomial deviates with small n, you can easily modify the code in `Binomialdev` to get multiple deviates ($\sim 64/n$, in fact) from each execution of the bit-parallel code.
- Do you need exact deviates, or would an approximation do? If your distribution of interest can be approximated by a normal distribution, consider substituting `Normaldev`, above, especially if you also code the uniform random generation inline.
- If you sum exactly 12 uniform deviates $U(0, 1)$ and then subtract 6, you get a pretty good approximation of a normal deviate $N(0, 1)$. This is definitely slower then `Normaldev` (not to mention less accurate) on a general-purpose CPU. However, there are reported to be some special-purpose signal processing chips in which all the operations can be done with integer arithmetic and in parallel.

See Gentle [3], Ripley [4], Devroye [6], Bratley [7], and Knuth [8] for many additional algorithms.

CITED REFERENCES AND FURTHER READING:

Kinderman, A.J. and Monahan, J.F 1977, "Computer Generation of Random Variables Using the Ratio of Uniform Deviates," *ACM Transactions on Mathematical Software*, vol. 3, pp. 257–260.[1]

Leva, J.L. 1992. "A Fast Normal Random Number Generator," *ACM Transactions on Mathematical Software*, vol. 18, no. 4, pp. 449-453.[2]

Gentle, J.E. 2003, *Random Number Generation and Monte Carlo Methods*, 2nd ed. (New York: Springer), Chapters 4–5.[3]

Ripley, B.D. 1987, *Stochastic Simulation* (New York: Wiley).[4]

Marsaglia, G. and Tsang W-W. 2000, "A Simple Method for Generating Gamma Variables," *ACM Transactions on Mathematical Software*, vol. 26, no. 3, pp. 363–372.[5]

Devroye, L. 1986, *Non-Uniform Random Variate Generation* (New York: Springer).[6]

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation*, 2nd ed. (New York: Springer).[7].

Knuth, D.E. 1997, *Seminumerical Algorithms*, 3rd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), pp. 125ff.[8]

7.4 Multivariate Normal Deviates

A multivariate random deviate of dimension M is a point in M-dimensional space. Its coordinates are a vector, each of whose M components are random — but not, in general, independently so, or identically distributed. The special case of *multivariate normal deviates* is defined by the multidimensional Gaussian density function

$$N(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{M/2} \det(\boldsymbol{\Sigma})^{1/2}} \exp[-\tfrac{1}{2}(\mathbf{x} - \boldsymbol{\mu}) \cdot \boldsymbol{\Sigma}^{-1} \cdot (\mathbf{x} - \boldsymbol{\mu})] \tag{7.4.1}$$

where the parameter $\boldsymbol{\mu}$ is a vector that is the mean of the distribution, and the parameter $\boldsymbol{\Sigma}$ is a symmetrical, positive-definite matrix that is the distribution's covariance.

There is a quite general way to construct a vector deviate $\mathbf{x}$ with a specified covariance $\boldsymbol{\Sigma}$ and mean $\boldsymbol{\mu}$, starting with a vector $\mathbf{y}$ of independent random deviates of zero mean and unit variance: First, use Cholesky decomposition (§2.9) to factor $\boldsymbol{\Sigma}$ into a left triangular matrix $\mathbf{L}$ times its transpose,

$$\boldsymbol{\Sigma} = \mathbf{L}\mathbf{L}^T \tag{7.4.2}$$

This is always possible because $\boldsymbol{\Sigma}$ is positive-definite, and you need do it only once for each distinct $\boldsymbol{\Sigma}$ of interest. Next, whenever you want a new deviate $\mathbf{x}$, fill $\mathbf{y}$ with independent deviates of unit variance and then construct

$$\mathbf{x} = \mathbf{L}\mathbf{y} + \boldsymbol{\mu} \tag{7.4.3}$$

The proof is straightforward, with angle brackets denoting expectation values: Since the components y_i are independent with unit variance, we have

$$\langle \mathbf{y} \otimes \mathbf{y} \rangle = \mathbf{1} \tag{7.4.4}$$

where $\mathbf{1}$ is the identity matrix. Then,

$$\begin{aligned} \langle (\mathbf{x} - \boldsymbol{\mu}) \otimes (\mathbf{x} - \boldsymbol{\mu}) \rangle &= \langle (\mathbf{L}\mathbf{y}) \otimes (\mathbf{L}\mathbf{y}) \rangle \\ &= \left\langle \mathbf{L}(\mathbf{y} \otimes \mathbf{y})\mathbf{L}^T \right\rangle = \mathbf{L} \langle \mathbf{y} \otimes \mathbf{y} \rangle \mathbf{L}^T \\ &= \mathbf{L}\mathbf{L}^T = \boldsymbol{\Sigma} \end{aligned} \tag{7.4.5}$$

As general as this procedure is, it is, however, rarely useful for anything except multivariate *normal* deviates. The reason is that while the components of $\mathbf{x}$ indeed have the right mean and covariance structure, their detailed distribution is not anything "nice." The x_i's are linear combinations of the y_i's, and, in general, a linear combination of random variables is distributed as a complicated convolution of their individual distributions.

For Gaussians, however, we do have "nice." All linear combinations of normal deviates are themselves normally distributed, and completely defined by their mean and covariance structure. Thus, if we always fill the components of $\mathbf{y}$ with normal deviates,

$$y_i \sim \mathrm{N}(0,1) \tag{7.4.6}$$

then the deviate (7.4.3) will be distributed according to equation (7.4.1).

Implementation is straightforward, since the `Cholesky` structure both accomplishes the decomposition and provides a method for doing the matrix multiplication efficiently, taking advantage of $\mathbf{L}$'s triangular structure. The generation of normal deviates is inline for efficiency, identical to `Normaldev` in §7.3.

multinormaldev.h

```
struct Multinormaldev : Ran {
```
Structure for multivariate normal deviates.
```
    Int mm;
    VecDoub mean;
    MatDoub var;
    Cholesky chol;
    VecDoub spt, pt;

    Multinormaldev(Ullong j, VecDoub &mmean, MatDoub &vvar) :
    Ran(j), mm(mmean.size()), mean(mmean), var(vvar), chol(var),
    spt(mm), pt(mm) {
```
    Constructor. Arguments are the random generator seed, the (vector) mean, and the (matrix) covariance. Cholesky decomposition of the covariance is done here.
```
        if (var.ncols() != mm || var.nrows() != mm) throw("bad sizes");
    }

    VecDoub &dev() {
```
    Return a multivariate normal deviate.
```
        Int i;
        Doub u,v,x,y,q;
        for (i=0;i<mm;i++) {                    Fill a vector of independent normal deviates.
            do {
                u = doub();
                v = 1.7156*(doub()-0.5);
                x = u - 0.449871;
                y = abs(v) + 0.386595;
                q = SQR(x) + y*(0.19600*y-0.25472*x);
            } while (q > 0.27597
                && (q > 0.27846 || SQR(v) > -4.*log(u)*SQR(u)));
            spt[i] = v/u;
        }
        chol.elmult(spt,pt);                    Apply equation (7.4.3).
        for (i=0;i<mm;i++) {pt[i] += mean[i];}
        return pt;
    }

};
```

7.4.1 Decorrelating Multiple Random Variables

Although not directly related to the generation of random deviates, this is a convenient place to point out how Cholesky decomposition can be used in the reverse manner, namely to find linear combinations of correlated random variables that have no correlation. In this application we are given a vector $\mathbf{x}$ whose components have a known covariance $\mathbf{\Sigma}$ and mean $\boldsymbol{\mu}$. Decomposing $\mathbf{\Sigma}$ as in equation (7.4.2), we assert

that

$$\mathbf{y} = \mathbf{L}^{-1}(\mathbf{x} - \boldsymbol{\mu}) \tag{7.4.7}$$

has uncorrelated components, each of unit variance. Proof:

$$\begin{aligned} \langle \mathbf{y} \otimes \mathbf{y} \rangle &= \left\langle (\mathbf{L}^{-1}[\mathbf{x} - \boldsymbol{\mu}]) \otimes (\mathbf{L}^{-1}[\mathbf{x} - \boldsymbol{\mu}]) \right\rangle \\ &= \mathbf{L}^{-1} \langle (\mathbf{x} - \boldsymbol{\mu}) \otimes (\mathbf{x} - \boldsymbol{\mu}) \rangle \mathbf{L}^{-1T} \\ &= \mathbf{L}^{-1} \boldsymbol{\Sigma} \mathbf{L}^{-1T} = \mathbf{L}^{-1} \mathbf{L} \mathbf{L}^T \mathbf{L}^{-1T} = \mathbf{1} \end{aligned} \tag{7.4.8}$$

Be aware that this linear combination is not unique. In fact, once you have obtained a vector $\mathbf{y}$ of uncorrelated components, you can perform any rotation on it and still have uncorrelated components. In particular, if $\mathbf{K}$ is an orthogonal matrix, so that

$$\mathbf{K}^T \mathbf{K} = \mathbf{K} \mathbf{K}^T = \mathbf{1} \tag{7.4.9}$$

then

$$\langle (\mathbf{K}\mathbf{y}) \otimes (\mathbf{K}\mathbf{y}) \rangle = \mathbf{K} \langle \mathbf{y} \otimes \mathbf{y} \rangle \mathbf{K}^T = \mathbf{K} \mathbf{K}^T = \mathbf{1} \tag{7.4.10}$$

A common (though slower) alternative to Cholesky decomposition is to use the Jacobi transformation (§11.1) to decompose $\boldsymbol{\Sigma}$ as

$$\boldsymbol{\Sigma} = \mathbf{V} \mathrm{diag}(\sigma_i^2) \mathbf{V}^T \tag{7.4.11}$$

where $\mathbf{V}$ is the orthogonal matrix of eigenvectors, and the σ_i's are the standard deviations of the (new) uncorrelated variables. Then $\mathbf{V}\mathrm{diag}(\sigma_i)$ plays the role of $\mathbf{L}$ in the proofs above.

Section §16.1.1 discusses some further applications of Cholesky decomposition relating to multivariate random variables.

7.5 Linear Feedback Shift Registers

A *linear feedback shift register* (LFSR) consists of a *state vector* and a certain kind of *update rule*. The state vector is often the set of bits in a 32- or 64-bit word, but it can sometimes be a set of words in an array. To qualify as an LFSR, the update rule must generate a *linear* combination of the bits (or words) in the current state, and then shift that result onto one end of the state vector. The oldest value, at the other end of the state vector, falls off and is gone. The output of an LFSR consists of the sequence of new bits (or words) as they are shifted in.

For single bits, "linear" means arithmetic modulo 2, which is the same as using the logical XOR operation for $+$ and the logical AND operation for $\times$. It is convenient, however, to write equations using the arithmetic notation. So, for an LFSR of length n, the words in the paragraph above translate to

$$\begin{aligned} a'_1 &= \left(\sum_{j=1}^{n-1} c_j a_j \right) + a_n \\ a'_i &= a_{i-1}, \qquad i = 2, \ldots, n \end{aligned} \tag{7.5.1}$$

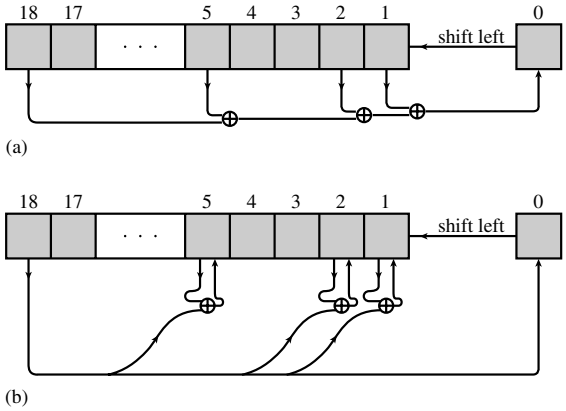

Figure 7.5.1. Two related methods for obtaining random bits from a shift register and a primitive polynomial modulo 2. (a) The contents of selected taps are combined by XOR (addition modulo 2), and the result is shifted in from the right. This method is easiest to implement in hardware. (b) Selected bits are modified by XOR with the leftmost bit, which is then shifted in from the right. This method is easiest to implement in software.

Here $\mathbf{a}'$ is the new state vector, derived from $\mathbf{a}$ by the update rule as shown. The reason for singling out a_n in the first line above is that its coefficient c_n must be $\equiv 1$. Otherwise, the LFSR wouldn't be of length n, but only of length up to the last nonzero coefficient in the c_j's.

There is also a reason for numbering the bits (henceforth we consider only the case of a vector of bits, not of words) starting with 1 rather than the more comfortable 0. The mathematical properties of equation (7.5.1) derive from the properties of the polynomials over the integers modulo 2. The polynomial associated with (7.5.1) is

$$P(x) = x^n + c_{n-1}x^{n-1} + \cdots + c_2x^2 + c_1x + 1 \tag{7.5.2}$$

where each of the c_i's has the value 0 or 1. So, c_0, like c_n, exists but is implicitly $\equiv 1$. There are several notations for describing specific polynomials like (7.5.2). One is to simply list the values i for which c_i is nonzero (by convention including c_n and c_0). So the polynomial

$$x^{18} + x^5 + x^2 + x + 1 \tag{7.5.3}$$

is abbreviated as

$$(18, 5, 2, 1, 0) \tag{7.5.4}$$

Another, when a value of n (here 18), and $c_n = c_0 = 1$, is assumed, is to construct a "serial number" from the binary word $c_{n-1}c_{n-1}\cdots c_2c_1$ (by convention now excluding c_n and c_0). For (7.5.3) this would be 19, that is, $2^4 + 2^1 + 2^0$. The nonzero c_i's are often referred to as an LFSR's *taps*.

Figure 7.5.1(a) illustrates how the polynomial (7.5.3) and (7.5.4) looks as an update process on a register of 18 bits. Bit 0 is the temporary where a bit that is to become the new bit 1 is computed.

The maximum period of an LFSR of n bits, before its output starts repeating, is $2^n - 1$. This is because the maximum number of distinct states is 2^n, but the special vector with all bits zero simply repeats itself with period 1. If you pick a random polynomial $P(x)$, then the generator you construct will usually not be full-period. A

fraction of polynomials over the integers modulo 2 are *irreducible*, meaning that they can't be factored. A fraction of the irreducible polynomials are *primitive*, meaning that they generate maximum period LFSRs. For example, the polynomial $x^2 + 1 = (x + 1)(x + 1)$ is not irreducible, so it is not primitive. (Remember to do arithmetic on the coefficients mod 2.) The polynomial $x^4 + x^3 + x^2 + x + 1$ is irreducible, but it turns out not to be primitive. The polynomial $x^4 + x + 1$ is both irreducible and primitive.

Maximum period LFSRs are often used as sources of random bits in hardware devices, because logic like that shown in Figure 7.5.1(a) requires only a few gates and can be made to run extremely fast. There is not much of a niche for LFSRs in software applications, because implementing equation (7.5.1) in code requires at least two full-word logical operations for each nonzero c_i, and all this work produces a meager one bit of output. We call this "Method I." A better software approach, "Method II," is not obviously an LFSR at all, but it turns out to be mathematically equivalent to one. It is shown in Figure 7.5.1(b). In code, this is implemented from a primitive polynomial as follows:

Let `maskp` and `maskn` be two bit masks,

$$\begin{array}{lcccccccc} \texttt{maskp} \equiv & (0 & \cdots & 0 & c_{n-1} & c_{n-2} & \cdots & c_2 & c_1) \\ \texttt{maskn} \equiv & (0 & \cdots & 1 & 0 & 0 & \cdots & 0 & 0) \end{array} \tag{7.5.5}$$

Then, a word **a** is updated by

```
if (a & maskn) a = ((a ^ maskp) << 1) | 1;
else a <<= 1;
```
(7.5.6)

You should work through the above prescription to see that it is identical to what is shown in the figure. The output of this update (still only one bit) can be taken as `(a & maskn)`, or for that matter any fixed bit in `a`.

LFSRs (either Method I or Method II) are sometimes used to get random m-bit words by concatenating the output bits from m consecutive updates (or, equivalently for Method I, grabbing the low-order m bits of state after every m updates). This is generally a bad idea, because the resulting words usually fail some standard statistical tests for randomness. It is especially a bad idea if m and $2^n - 1$ are not relatively prime, in which case the method does not even give all m-bit words uniformly.

Next, we'll develop a bit of theory to see the relation between Method I and Method II, and this will lead us to a routine for testing whether any given polynomial (expressed as a bit string of c_i's) is primitive. But, for now, if you only need a table of some primitive polynomials go get going, one is provided on the next page.

Since the update rule (7.5.1) is linear, it can be written as a matrix $\mathbf{M}$ that multiplies from the left a column vector of bits **a** to produce an updated state $\mathbf{a}'$. (Note that the low-order bits of **a** start at the top of the column vector.) One can readily read off

$$\mathbf{M} = \begin{bmatrix} c_1 & c_2 & \ldots & c_{n-2} & c_{n-1} & 1 \\ 1 & 0 & \ldots & 0 & 0 & 0 \\ 0 & 1 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & 0 & 0 \\ 0 & 0 & \ldots & 0 & 1 & 0 \end{bmatrix} \tag{7.5.7}$$

Some Primitive Polynomials Modulo 2 (after Watson [1])

(1, 0)	(51, 6, 3, 1, 0)
(2, 1, 0)	(52, 3, 0)
(3, 1, 0)	(53, 6, 2, 1, 0)
(4, 1, 0)	(54, 6, 5, 4, 3, 2, 0)
(5, 2, 0)	(55, 6, 2, 1, 0)
(6, 1, 0)	(56, 7, 4, 2, 0)
(7, 1, 0)	(57, 5, 3, 2, 0)
(8, 4, 3, 2, 0)	(58, 6, 5, 1, 0)
(9, 4, 0)	(59, 6, 5, 4, 3, 1, 0)
(10, 3, 0)	(60, 1, 0)
(11, 2, 0)	(61, 5, 2, 1, 0)
(12, 6, 4, 1, 0)	(62, 6, 5, 3, 0)
(13, 4, 3, 1, 0)	(63, 1, 0)
(14, 5, 3, 1, 0)	(64, 4, 3, 1, 0)
(15, 1, 0)	(65, 4, 3, 1, 0)
(16, 5, 3, 2, 0)	(66, 8, 6, 5, 3, 2, 0)
(17, 3, 0)	(67, 5, 2, 1, 0)
(18, 5, 2, 1, 0)	(68, 7, 5, 1, 0)
(19, 5, 2, 1, 0)	(69, 6, 5, 2, 0)
(20, 3, 0)	(70, 5, 3, 1, 0)
(21, 2, 0)	(71, 5, 3, 1, 0)
(22, 1, 0)	(72, 6, 4, 3, 2, 1, 0)
(23, 5, 0)	(73, 4, 3, 2, 0)
(24, 4, 3, 1, 0)	(74, 7, 4, 3, 0)
(25, 3, 0)	(75, 6, 3, 1, 0)
(26, 6, 2, 1, 0)	(76, 5, 4, 2, 0)
(27, 5, 2, 1, 0)	(77, 6, 5, 2, 0)
(28, 3, 0)	(78, 7, 2, 1, 0)
(29, 2, 0)	(79, 4, 3, 2, 0)
(30, 6, 4, 1, 0)	(80, 7, 5, 3, 2, 1, 0)
(31, 3, 0)	(81, 4 0)
(32, 7, 5, 3, 2, 1, 0)	(82, 8, 7, 6, 4, 1, 0)
(33, 6, 4, 1, 0)	(83, 7, 4, 2, 0)
(34, 7, 6, 5, 2, 1, 0)	(84, 8, 7, 5, 3, 1, 0)
(35, 2, 0)	(85, 8, 2, 1, 0)
(36, 6, 5, 4, 2, 1, 0)	(86, 6, 5, 2, 0)
(37, 5, 4, 3, 2, 1, 0)	(87, 7, 5, 1, 0)
(38, 6, 5, 1, 0)	(88, 8, 5, 4, 3, 1, 0)
(39, 4, 0)	(89, 6, 5, 3, 0)
(40, 5, 4 3, 0)	(90, 5, 3, 2, 0)
(41, 3, 0)	(91, 7, 6, 5, 3, 2, 0)
(42, 5, 4, 3, 2, 1, 0)	(92, 6, 5, 2, 0)
(43, 6, 4, 3, 0)	(93, 2, 0)
(44, 6, 5, 2, 0)	(94, 6, 5, 1, 0)
(45, 4, 3, 1, 0)	(95, 6, 5, 4, 2, 1, 0)
(46, 8, 5, 3, 2, 1, 0)	(96, 7, 6, 4, 3, 2, 0)
(47, 5, 0)	(97, 6, 0)
(48, 7, 5, 4, 2, 1, 0)	(98, 7, 4, 3, 2, 1, 0)
(49, 6, 5, 4, 0)	(99, 7, 5, 4, 0)
(50, 4, 3, 2, 0)	(100, 8, 7, 2, 0)

What are the conditions on $\mathbf{M}$ that give a full-period generator, and thereby prove that the polynomial with coefficients c_i is primitive? Evidently we must have

$$\mathbf{M}^{(2^n-1)} = \mathbf{1} \tag{7.5.8}$$

where $\mathbf{1}$ is the identity matrix. This states that the period, or some multiple of it, is $2^n - 1$. But the only possible such multiples are integers that divide $2^n - 1$. To rule these out, and ensure a full period, we need only check that

$$\mathbf{M}^{q_k} \neq \mathbf{1}, \qquad q_k \equiv (2^n - 1)/f_k \tag{7.5.9}$$

for every prime factor f_k of $2^n - 1$. (This is exactly the logic behind the tests of the matrix $\mathbf{T}$ that we described, but did not justify, in §7.1.2.)

It may at first sight seem daunting to compute the humongous powers of $\mathbf{M}$ in equations (7.5.8) and (7.5.9). But, by the method of repeated squaring of $\mathbf{M}$, each such power takes only about n (a number like 32 or 64) matrix multiplies. And, since all the arithmetic is done modulo 2, there is no possibility of overflow! The conditions (7.5.8) and (7.5.9) are in fact an efficient way to test a polynomial for primitiveness. The following code implements the test. Note that you must customize the constants in the constructor for your choice of n (called `N` in the code), in particular the prime factors of $2^n - 1$. The case $n = 32$ is shown. Other than that customization, the code as written is valid for $n \leq 64$. The input to the test is the "serial number," as defined above following equation (7.5.4), of the polynomial to be tested. After declaring an instance of the `Primpolytest` structure, you can repeatedly call its `test()` method to test multiple polynomials. To make `Primpolytest` entirely self-contained, matrices are implemented as linear arrays, and the structure builds from scratch the few matrix operations that it needs. This is inelegant, but effective.

primpolytest.h

```
struct Primpolytest {
Test polynomials over the integers mod 2 for primitiveness.
    Int N, nfactors;
    VecUllong factors;
    VecInt t,a,p;

    Primpolytest() : N(32), nfactors(5), factors(nfactors), t(N*N),
        a(N*N), p(N*N) {
        Constructor. The constants are specific to 32-bit LFSRs.
        Ullong factordata[5] = {3,5,17,257,65537};
        for (Int i=0;i<nfactors;i++) factors[i] = factordata[i];
    }

    Int ispident() {                          Utility to test if p is the identity matrix.
        Int i,j;
        for (i=0; i<N; i++) for (j=0; j<N; j++) {
            if (i == j) { if (p[i*N+j] != 1) return 0; }
            else {if (p[i*N+j] != 0) return 0; }
        }
        return 1;
    }

    void mattimeseq(VecInt &a, VecInt &b) {   Utility for a *= b on matrices a and b.
        Int i,j,k,sum;
        VecInt tmp(N*N);
        for (i=0; i<N; i++) for (j=0; j<N; j++) {
            sum = 0;
            for (k=0; k<N; k++) sum += a[i*N+k] * b[k*N+j];
            tmp[i*N+j] = sum & 1;
        }
        for (k=0; k<N*N; k++) a[k] = tmp[k];
    }

    void matpow(Ullong n) {                   Utility for matrix p = a^n by successive
        Int k;                                    squares.
```

```
        for (k=0; k<N*N; k++) p[k] = 0;
        for (k=0; k<N; k++) p[k*N+k] = 1;
        while (1) {
            if (n & 1) mattimeseq(p,a);
            n >>= 1;
            if (n == 0) break;
            mattimeseq(a,a);
        }
    }

    Int test(Ullong n) {
    Main test routine. Returns 1 if the polynomial with serial number n (see text) is primitive,
    0 otherwise.
        Int i,k,j;
        Ullong pow, tnm1, nn = n;
        tnm1 = ((Ullong)1 << N) - 1;
        if (n > (tnm1 >> 1)) throw("not a polynomial of degree N");
        for (k=0; k<N*N; k++) t[k] = 0;          Construct the update matrix in t.
        for (i=1; i<N; i++) t[i*N+(i-1)] = 1;
        j=0;
        while (nn) {
            if (nn & 1) t[j] = 1;
            nn >>= 1;
            j++;
        }
        t[N-1] = 1;
        for (k=0; k<N*N; k++) a[k] = t[k];       Test that t^tnm1 is the identity matrix.
        matpow(tnm1);
        if (ispident() != 1) return 0;
        for (i=0; i<nfactors; i++) {             Test that the t to the required submulti-
            pow = tnm1/factors[i];                   ple powers is not the identity matrix.
            for (k=0; k<N*N; k++) a[k] = t[k];
            matpow(pow);
            if (ispident() == 1) return 0;
        }
        return 1;
    }
};
```

It is straightforward to generalize this method to $n > 64$ or to prime moduli p other than 2. If $p^n > 2^{64}$, you'll need a multiword binary representation of the integers $p^n - 1$ and its quotients with its prime factors, so that `matpow` can still find powers by successive squares. Note that the computation time scales roughly as $O(n^4)$, so $n = 64$ is fast, while $n = 1024$ would be rather a long calculation.

Some random primitive polynomials for $n = 32$ bits (giving their serial numbers as decimal values) are 2046052277, 1186898897, 221421833, 55334070, 1225518245, 216563424, 1532859853, 1735381519, 2049267032, 1363072601, and 130420448. Some random ones for $n = 64$ bits are 926773948609480634, 3195735403700392248, 4407129700254524327, 256457582706860311, 5017679982664373343, and 1723461400905116882.

Given a matrix $\mathbf{M}$ that satisfies equations (7.5.8) and (7.5.9), there are some related matrices that also satisfy those relations. An example is the inverse of $\mathbf{M}$, which you can easily verify as

$$\mathbf{M}^{-1} = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & 1 \\ 1 & c_1 & c_2 & \dots & c_{n-2} & c_{n-1} \end{bmatrix} \tag{7.5.10}$$

This is the update rule that backs up a state $\mathbf{a}'$ to its predecessor state $\mathbf{a}$. You can easily convert (7.5.10) to a prescription analogous to equation (7.5.1) or to Figure 7.5.1(a).

Another matrix satisfying the relations that guarantee a full period is the transpose of the

inverse (or inverse of the transpose) of $\mathbf{M}$,

$$\left(\mathbf{M}^{-1}\right)^T = \begin{bmatrix} 0 & 0 & \dots & 0 & 0 & 1 \\ 1 & 0 & \dots & 0 & 0 & c_1 \\ 0 & 1 & \dots & 0 & 0 & c_2 \\ \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 & c_{n-2} \\ 0 & 0 & \dots & 0 & 1 & c_{n-1} \end{bmatrix} \tag{7.5.11}$$

Surprise! This is exactly Method II, as also shown in Figure 7.5.1(b). (Work it out.)

Even more specifically, the sequence of bits output by a Method II LFSR based on a primitive polynomial $P(x)$ is identical to the sequence output by a Method I LFSR that uses the *reciprocal polynomial* $x^n P(1/x)$. The proof is a bit beyond our scope, but it is essentially because the matrix $\mathbf{M}$ and its transpose are both roots of the characteristic polynomial, equation (7.5.2), while the inverse matrix $\mathbf{M}^{-1}$ and its transpose are both roots of the reciprocal polynomial. The reciprocal polynomial, as you can easily check from the definition, just swaps the positions of nonzero coefficients end-to-end. For example, the reciprocal polynomial of equation (7.5.3) is $(18, 17, 16, 13, 1)$. If a polynomial is primitive, so is its reciprocal.

Try this experiment: Run a Method II generator for a while. Then take n consecutive bits of its output (from its highest bit, say) and put them into a Method I shift register as initialization (low bit the most recent one). Now step the two methods together, using the reciprocal polynomial in the Method I. You'll get identical output from the two generators.

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), pp. 30ff.

Horowitz, P., and Hill, W. 1989, *The Art of Electronics*, 2nd ed. (Cambridge, UK: Cambridge University Press), §9.32 – §9.37.

Tausworthe, R.C. 1965, "Random Numbers Generated by Linear Recurrence Modulo Two," *Mathematics of Computation*, vol. 19, pp. 201–209.

Watson, E.J. 1962, "Primitive Polynomials (Mod 2)," *Mathematics of Computation*, vol. 16, pp. 368–369.[1]

7.6 Hash Tables and Hash Memories

It's a strange dream. You're in a kind of mailroom whose walls are lined with numbered pigeonhole boxes. A man, Mr. Hacher, sits at a table. You are standing. There is an in-basket mounted on the wall. Your job is to take letters from the in-basket and sort them into the pigeonholes.

But how? The letters are addressed by name, while the pigeonholes are only numbered. That is where Mr. Hacher comes in. You show him each letter, and he immediately tells you its pigeonhole number. He always gives the same number for the same name, while different names always get different numbers (and therefore unique pigeonholes).

Over time, as the number of addressees grows, there are fewer and fewer empty boxes until, finally, none at all. This is not a problem as long as letters arrive only for existing boxholders. But one day, you spot a new name on an envelope. With trepidation you put it in front of Mr. Hacher . . . and you wake up!

Mr. Hacher and his table are a *hash table*. A hash table behaves as if it keeps a running ledger of all the *hash keys* (the addressee names) that it has ever seen, assigns a unique number to each, and is able to look through all the names for every new query, either returning the same number as before (for a repeat key) or, for a new key, assigning a new one. There is usually also an option to erase a key.

The goal in implementing a hash table is to make all these functions take only a few computer operations each, not even $O(\log N)$. That is quite a trick, if you think about it. Even if you somehow maintain an ordered or alphabetized list of keys, it will still take $O(\log N)$ operations to find a place in the list, by bisection, say. The big idea behind hash tables is the use of random number techniques (§7.1) to map a hash key to a pseudo-random integer between 0 and $N-1$, where N is the total number of pigeonholes. Here we definitely want pseudo-random and not random integers, because the same key must produce the same integer each time.

In first approximation, ideally much of the time, that initial pseudo-random integer, called the output of the *hash function*, or (for short) the key's *hash*, is what the hash table puts out, i.e., the number given out by Mr. Hacher. However, it is possible that, by chance, two keys have the same hash; in fact this becomes increasingly probable as the number of distinct keys approaches N, and a certainty when N is exceeded (the *pigeonhole principle*). The implementation of a hash table therefore requires a *collision strategy* that ensures that unique integers are returned, even for (different) keys that have the same hash.

Many vendors' implementations of the C++'s Standard Template Library (STL) provide a hash table as the class `hash_map`. Unfortunately, at this writing, `hash_map` is not a part of the actual STL standard, and the quality of vendor implementations is also quite variable. We therefore here implement our own; thereby we can both learn more about the principles involved and build in some specific features that will be useful later in this book (for example §21.8 and §21.6).

7.6.1 Hash Function Object

By a hash function object we mean a structure that combines a hashing algorithm (as in §7.1) with the "glue" needed to make a hash table. The object should map an arbitrary key type `keyT`, which itself may be a structure containing multiple data values, into (for our implementation) a pseudo-random 64-bit integer. All the hash function object really needs to know about `keyT` is its length in bytes, that is, `sizeof(keyT)`, since it doesn't care how those bytes are used, only that they are part of the key to be hashed. We therefore give the hash function object a constructor that tells it how many bytes to hash; and we let it access a key by a `void` pointer to the key's address. Thus the object can access those bytes any way it wants.

As a first example of a hash function object, let's just put a wrapper around the hash function algorithm of §7.1.4. This is quite efficient when `sizeof(keyT)` $= 4$ or 8.

hash.h

```
struct Hashfn1 {
```
Example of an object encapsulating a hash function for use by the class Hashmap.

```
    Ranhash hasher;                          The actual hash function.
    Int n;                                   Size of key in bytes.
    Hashfn1(Int nn) : n(nn) {}               Constructor just saves key size.
    Ullong fn(const void *key) {             Function that returns hash from key.
        Uint *k;
        Ullong *kk;
```

```
        switch (n) {
            case 4:
                k = (Uint *)key;
                return hasher.int64(*k);          Return 64-bit hash of 32-bit key.
            case 8:
                kk = (Ullong *)key;\
                return hasher.int64(*kk);         Return 64-bit hash of 64-bit key.
            default:
                throw("Hashfn1 is for 4 or 8 byte keys only.");
        }
    }
};
```

(Since n is constant for the life of the object, it's a bit inefficient to be testing it on every call; you should edit out the unnecessary code when you know n in advance.)

More generally, a hash function object can be designed to work on arbitrary sized keys by incorporating them into a final hash value a byte at a time. There is a trade-off between speed and degree-of-randomness. Historically, hash functions have favored speed, with simple incorporation rules like

$$\begin{aligned} h_0 &= \text{some fixed constant} \\ h_i &= (m\ h_{i-1}\ \text{op}\ k_i) \mod 2^{32} \qquad (i = 1 \ldots K) \end{aligned} \tag{7.6.1}$$

Here k_i is the ith byte of the key ($1 \le i \le K$), m is a multiplier with popular values that include 33, 63689, and $2^{16} + 2^6 - 1$ (doing the multiplication by shifts and adds in the first and third cases), and "op" is either addition or bitwise XOR. You get the mod function for free when you use 32-bit unsigned integer arithmetic. However, since 64-bit arithmetic is fast on modern machines, we think that the days of small multipliers, or many operations changing only a few bits at a time, are over. We favor hash functions that can pass good tests for randomness. (When you know a lot about your keys, it is possible to design hash functions that are even *better* than random, but that is beyond our scope here.)

A hash function object may also do some initialization (of tables, etc.) when it is created. Unlike a random number generator, however, it may not store any history-dependent state between calls, because it must return the same hash for the same key every time. Here is an example of a self-contained hash function object for keys of any length. This is the hash function object that we will use below.

hash.h

```
struct Hashfn2 {
```

Another example of an object encapsulating a hash function, allowing arbitrary fixed key sizes or variable-length null terminated strings. The hash function algorithm is self-contained.

```
    static Ullong hashfn_tab[256];
    Ullong h;
    Int n;
    Hashfn2(Int nn) : n(nn) {
        if (n == 1) n = 0;
        h = 0x544B2FBACAAF1684LL;
        for (Int j=0; j<256; j++) {
            for (Int i=0; i<31; i++) {
                h = (h >>  7) ^ h;
                h = (h << 11) ^ h;
                h = (h >> 10) ^ h;
            }
            hashfn_tab[j] = h;
        }
    }
```

Size of key in bytes, when fixed size.

Null terminated string key signaled by $n = 0$ or 1.

Length 256 lookup table is initialized with values from a 64-bit Marsaglia generator stepped 31 times between each.

```
    Ullong fn(const void *key) {            Function that returns hash from key.
        Int j;
        char *k = (char *)key;              Cast the key pointer to char pointer.
        h=0xBB40E64DA205B064LL;
        j=0;
        while (n ? j++ < n : *k) {          Fixed length or else until null.
            h = (h * 7664345821815920749LL) ^ hashfn_tab[(unsigned char)(*k)];
            k++;
        }
        return h;
    }
};
Ullong Hashfn2::hashfn_tab[256];            Defines storage for the lookup table.
```

The method used is basically equation (7.6.1), but (i) with a large constant that is known to be a good multiplier for a linear congruential random number generator mod 2^{64}, and, more importantly, (ii) a table lookup that substitutes a random (but fixed) 64-bit value for every byte value in $0 \ldots 255$. Note also the tweak that allows `Hashfn2` to be used either for fixed length key types (call constructor with `n` > 1) or with null terminated byte arrays of variable length (call constructor with `n` $= 0$ or 1).

7.6.2 Hash Table

By *hash table* we mean an object with the functionality of Mr. Hacher (and his table) in the dream, namely to turn arbitrary keys into unique integers in a specified range. Let's dive right in. In outline, the `Hashtable` object is

```
template<class keyT, class hfnT> struct Hashtable {                    hash.h
```
Instantiate a hash table, with methods for maintaining a one-to-one correspondence between arbitrary keys and unique integers in a specified range.
```
    Int nhash, nmax, nn, ng;
    VecInt htable, next, garbg;
    VecUllong thehash;
    hfnT hash;                          An instance of a hash function object.
    Hashtable(Int nh, Int nv);
```
Constructor. Arguments are size of hash table and max number of stored elements (keys).
```
    Int iget(const keyT &key);          Return integer for a previously set key.
    Int iset(const keyT &key);          Return unique integer for a new key.
    Int ierase(const keyT &key);        Erase a key.
    Int ireserve();                     Reserve an integer (with no key).
    Int irelinquish(Int k);             Un-reserve an integer.
};

template<class keyT, class hfnT>
Hashtable<keyT,hfnT>::Hashtable(Int nh, Int nv):
```
Constructor. Set `nhash`, the size of the hash table, and `nmax`, the maximum number of elements (keys) that can be accommodated. Allocate arrays appropriately.
```
    hash(sizeof(keyT)), nhash(nh), nmax(nv), nn(0), ng(0),
    htable(nh), next(nv), garbg(nv), thehash(nv) {
    for (Int j=0; j<nh; j++) { htable[j] = -1; }     Signifies empty.
}
```

A `Hashtable` object is templated by two class names: the class of the key (which may be as simple as `int` or as complicated as a multiply derived class) and the class of the hash function object (e.g., `Hashfn1` or `Hashfn2`, above). Note how the hash function object is automatically created using the size of `keyT`, so the user is not responsible for knowing this value. If you are going to use variable length, null

terminated byte arrays as keys, then the type of `keyT` is `char`, not `char*`; see §7.6.5 for an example.

The hash table object is created from two integer parameters. The most important one is `nm`, the maximum number of objects that can be stored — in the dream, the number of pigeonholes in the room. For now, suppose that the second parameter, `nh`, has the same value as `nm`.

The overall scheme is to convert arbitrary keys into integers in the range 0 ... `nh-1` that index into the array `htable`, by taking the output of the hash function modulo `nh`. That array's indexed element contains either -1, meaning "empty," or else an index in the range 0 ... `nm-1` that points into the arrays `thehash` and `next`. (For a computer science flavor one could do this with list elements linked by pointers, but in the spirit of numerical computation, we will use arrays; both ways are about equally efficient.)

An element in `thehash` contains the 64-bit hash of whatever key was previously assigned to that index. We will take the identity of two hashes as being positive proof that their keys were identical. Of course this is not really true. There is a probability of $2^{-64} \sim 5 \times 10^{-20}$ of two keys giving identical hashes by chance. To guarantee error-free performance, a hash table must in fact store the actual key, not just the hash; but for our purposes we will accept the very small chance that two elements might get confused. (Don't use these routines if you are typically storing more than a billion elements in a single hash table. But you already knew that!)

This 10^{-20} coincidence is *not* what is meant by *hash collision*. Rather, hash collisions occur when two hashes yield the same value modulo `nh`, so that they point to the same element in `htable`. That is not at all unusual, and we must provide for handling it. Elements in the array `next` contain values that index back into `thehash` and `next`, i.e., form a linked list. So, when two or more keys have landed on the same value i, $0 \le i <$ `nh`, and we want to retrieve a particular one of them, it will either be in the location `thehash`[i], or else in the (hopefully short) list that starts there and is linked by `next`[i], `next`[`next`[i]], and so forth.

We can now say more about the value that should be initially specified for the parameter `nh`. For a full table with all `nm` values assigned, the linked lists attached to each element of `htable` have lengths that are Poisson distributed with a mean $\lambda \equiv$ `nm`/`nh`. Thus, large λ (`nh` too small) implies a lot of list traversal, while small λ (`nh` too large) implies wasted space in `htable`. Conventional wisdom is to choose $\lambda \sim 0.75$, in which case (assuming a good hash function) 47% of `htable` will be empty, 67% of the nonempty elements will have lists of length one (i.e., you get the correct key on the first try), and the mean number of indirections (steps in traversing the `next` pointers) is 0.42. For $\lambda = 1$, that is, `nh` = `nm`, the values are 37% table empty, 58% first try hits, and 0.58 mean indirections. So, in this general range, any choice is basically fine. The general formulas are

$$\begin{aligned} \text{empty fraction} &= P_\lambda(0) = e^{-\lambda} \\ \text{first try hits} &= P_\lambda(1)/[1 - P_\lambda(0)] = \frac{\lambda e^{-\lambda}}{1 - e^{-\lambda}} \\ \text{mean indirections} &= \sum_{j=2}^{\infty} \frac{(j-1)P_\lambda(j)}{1 - P_\lambda(0)} = \frac{e^{-\lambda} - 1 + \lambda}{1 - e^{-\lambda}} \end{aligned} \tag{7.6.2}$$

where $P_\lambda(j)$ is the Poisson probability function.

Now to the implementations within `Hashtable`. The simplest to understand is the "get" function, which returns an index value only if the key was previously "set," and returns -1 (by convention) if it was not. Our data structure is designed to make this as fast as possible.

hash.h

```
template<class keyT, class hfnT>
Int Hashtable<keyT,hfnT>::iget(const keyT &key) {
```
Returns integer in `0..nmax-1` corresponding to key, or -1 if no such key was previously stored.
```
    Int j,k;
    Ullong pp = hash.fn(&key);                    Get 64-bit hash
    j = (Int)(pp % nhash);                        and map it into the hash table.
    for (k = htable[j]; k != -1; k = next[k]) {   Traverse linked list until an ex-
        if (thehash[k] == pp) {                       act match is found.
            return k;
        }
    }
    return -1;                                    Key was not previously stored.
}
```

A language subtlety to be noted is that `iget` receives `key` as a `const` reference, and then passes its *address*, namely `&key`, to the hash function object. C++ allows this, because the hash function object's `void` pointer argument is itself declared as `const`.

The routine that "sets" a key is slightly more complicated. If the key has previously been set, we want to return the same value as the first time. If it hasn't been set, we initialize the necessary links for the future.

hash.h

```
template<class keyT, class hfnT>
Int Hashtable<keyT,hfnT>::iset(const keyT &key) {
```
Returns integer in `0..nmax-1` that will henceforth correspond to `key`. If `key` was previously set, return the same integer as before.
```
    Int j,k,kprev;
    Ullong pp = hash.fn(&key);                Get 64-bit hash
    j = (Int)(pp % nhash);                    and map it into the hash table.
    if (htable[j] == -1) {                    Key not in table. Find a free integer, either
        k = ng ? garbg[--ng] : nn++ ;             new or previously erased.
        htable[j] = k;
    } else {                                  Key might be in table. Traverse list.
        for (k = htable[j]; k != -1; k = next[k]) {
            if (thehash[k] == pp) {
                return k;                     Yes. Return previous value.
            }
            kprev = k;
        }
        k = ng ? garbg[--ng] : nn++ ;         No. Get new integer.
        next[kprev] = k;
    }
    if (k >= nmax) throw("storing too many values");
    thehash[k] = pp;                          Store the key at the new or previous integer.
    next[k] = -1;
    return k;
}
```

A word here about garbage collection. When a key is erased (by the routine immediately below), we want to make its integer available to future "sets," so that `nmax` keys can always be stored. This is very easy to implement if we allocate a garbage array (`garbg`) and use it as a last-in first-out stack of available integers. The `set` routine above always checks this stack when it needs a new integer. (By the way, had we designed `Hashtable` with list elements linked by pointers, instead of

arrays, efficient garbage collection would have been more difficult to implement; see Stroustrop [1].)

hash.h
```
template<class keyT, class hfnT>
Int Hashtable<keyT,hfnT>::ierase(const keyT &key) {
```
Erase a key, returning the integer in 0..nmax-1 erased, or −1 if the key was not previously set.
```
    Int j,k,kprev;
    Ullong pp = hash.fn(&key);
    j = (Int)(pp % nhash);
    if (htable[j] == -1) return -1;      Key not previously set.
    kprev = -1;
    for (k = htable[j]; k != -1; k = next[k]) {
        if (thehash[k] == pp) {          Found key. Splice linked list around it.
            if (kprev == -1) htable[j] = next[k];
            else next[kprev] = next[k];
            garbg[ng++] = k;             Add k to garbage stack as an available integer.
            return k;
        }
        kprev = k;
    }
    return -1;                           Key not previously set.
}
```

Finally, `Hashtable` has routines that reserve and relinquish integers in the range 0 to `nmax`. When an integer is reserved, it is guaranteed not to be used by the hash table. Below, we'll use this feature as a convenience in constructing a hash memory that can store more than one element under a single key.

hash.h
```
template<class keyT, class hfnT>
Int Hashtable<keyT,hfnT>::ireserve() {
```
Reserve an integer in 0..nmax-1 so that it will not be used by set(), and return its value.
```
    Int k = ng ? garbg[--ng] : nn++ ;
    if (k >= nmax) throw("reserving too many values");
    next[k] = -2;
    return k;
}

template<class keyT, class hfnT>
Int Hashtable<keyT,hfnT>::irelinquish(Int k) {
```
Return to the pool an index previously reserved by reserve(), and return it, or return −1 if it was not previously reserved.
```
    if (next[k] != -2) {return -1;}
    garbg[ng++] = k;
    return k;
}
```

7.6.3 Hash Memory

The `Hashtable` class, above, implements Mr. Hacher's task. Building on it, we next implement *your* job in the dream, namely to do the actual storage and retrieval of arbitrary objects by arbitrary keys. This is termed a *hash memory*.

When you store into an ordinary computer memory, the value of anything previously stored there is overwritten. If you want your hash memory to behave the same way, then a hash memory class, `Hash`, derived from `Hashtable`, is almost trivial to write. The class is templated by three structure types: `keyT` for the key type; `elT` for the type of the element that is stored in the hash memory; and `hfnT`, as before, for the object that encapsulates the hash function of your choice.

hash.h

```
template<class keyT, class elT, class hfnT>
struct Hash : Hashtable<keyT, hfnT> {
```
Extend the `Hashtable` class with storage for elements of type `elT`, and provide methods for storing, retrieving. and erasing elements. `key` is passed by address in all methods.
```
    using Hashtable<keyT,hfnT>::iget;
    using Hashtable<keyT,hfnT>::iset;
    using Hashtable<keyT,hfnT>::ierase;
    vector<elT> els;

    Hash(Int nh, Int nm) : Hashtable<keyT, hfnT>(nh, nm), els(nm) {}
```
Same constructor syntax as `Hashtable`.
```
    void set(const keyT &key, const elT &el)
```
Store an element `el`.
```
        {els[iset(key)] = el;}

    Int get(const keyT &key, elT &el) {
```
Retrieve an element into `el`. Returns 0 if no element is stored under `key`, or 1 for success.
```
        Int ll = iget(key);
        if (ll < 0) return 0;
        el = els[ll];
        return 1;
    }

    elT& operator[] (const keyT &key) {
```
Store or retrieve an element using subscript notation for its key. Returns a reference that can be used as an l-value.
```
        Int ll = iget(key);
        if (ll < 0) {
            ll = iset(key);
            els[ll] = elT();
        }
        return els[ll];
    }

    Int count(const keyT &key) {
```
Return the number of elements stored under `key`, that is, either 0 or 1.
```
        Int ll = iget(key);
        return (ll < 0 ? 0 : 1);
    }

    Int erase(const keyT &key) {
```
Erase an element. Returns 1 for success, or 0 if no element is stored under `key`.
```
        return (ierase(key) < 0 ? 0 : 1);
    }
};
```

The `operator[]` method, above, is intended for two distinct uses. First, it implements an intuitive syntax for storing and retrieving elements, e.g.,

`myhash[` *some-key* `]` = *rhs*

for storing, and

lhs = `myhash[` *some-key* `]`

for retrieving. Note, however, that a small inefficiency is introduced, namely a superfluous call to `get` when an element is set for the first time. Second, the method returns a non-`const` reference that cannot only be used as an l-value, but also be pointed to, as in

some-pointer = &myhash[*some-key*]

Now the stored element can be referenced through the pointer, possibly multiple times, without any additional overhead of key lookup. This can be an important gain in efficiency in some applications. Of course you can also use the `set` and `get` methods directly.

7.6.4 Hash Multimap Memory

Next turn to the case where you want to be able to store *more than one* element under the same key. If ordinary computer memory behaved this way, you could set a variable to a series of values and have it remember all of them! Obviously this is a somewhat more complicated an extension of `Hashtable` than was `Hash`. We will call it `Mhash`, where the M stands for "multivalued" or "multimap." One requirement is to provide a convenient syntax for retrieving multiple values of a single key, one at a time. We do this by the functions `getinit` and `getnext`. Also, in `Mhash`, below, `nmax` now means the maximum number of *values* that can be stored, not the number of keys, which may in general be smaller.

The code, with comments, should be understandable without much additional explanation. We use the `reserve` and `relinquish` features of `Hashtable` so as to have a common numbering system for all stored elements, both the first instance of a key (which `Hashtable` must know about) and subsequent instances of the same key (which are invisible to `Hashtable` but managed by `Mhash` through the linked list `nextsis`).

hash.h

```
template<class keyT, class elT, class hfnT>
struct Mhash : Hashtable<keyT,hfnT> {
```
Extend the `Hashtable` class with storage for elements of type `elT`, allowing more than one element to be stored under a single key.
```
    using Hashtable<keyT,hfnT>::iget;
    using Hashtable<keyT,hfnT>::iset;
    using Hashtable<keyT,hfnT>::ierase;
    using Hashtable<keyT,hfnT>::ireserve;
    using Hashtable<keyT,hfnT>::irelinquish;
    vector<elT> els;
    VecInt nextsis;                          Links to next sister element under a single key.
    Int nextget;
    Mhash(Int nh, Int nm);                   Same constructor syntax as Hashtable.
    Int store(const keyT &key, const elT &el);    Store an element under key.
    Int erase(const keyT &key, const elT &el);    Erase a specified element under key.
    Int count(const keyT &key);              Count elements stored under key.
    Int getinit(const keyT &key);            Prepare to retrieve elements from key.
    Int getnext(elT &el);                    Retrieve next element specified by getinit.
};

template<class keyT, class elT, class hfnT>
Mhash<keyT,elT,hfnT>::Mhash(Int nh, Int nm)
    : Hashtable<keyT, hfnT>(nh, nm), nextget(-1), els(nm), nextsis(nm) {
    for (Int j=0; j<nm; j++) {nextsis[j] = -2;}   Initialize to "empty".
}

template<class keyT, class elT, class hfnT>
Int Mhash<keyT,elT,hfnT>::store(const keyT &key, const elT &el) {
```
Store an element `el` under `key`. Return index in `0..nmax-1`, giving the storage location utilized.
```
    Int j,k;
    j = iset(key);                           Find root index for this key.
    if (nextsis[j] == -2) {                  It is the first object with this key.
```

```
        els[j] = el;
        nextsis[j] = -1;                    −1 means it is the terminal element.
        return j;
    } else {
        while (nextsis[j] != -1) {j = nextsis[j];}    Traverse the tree.
        k = ireserve();                     Get a new index and link it into the list.
        els[k] = el;
        nextsis[j] = k;
        nextsis[k] = -1;
        return k;
    }
}
```

```
template<class keyT, class elT, class hfnT>
Int Mhash<keyT,elT,hfnT>::erase(const keyT &key, const elT &el) {
```

Erase an element `el` previously stored under `key`. Return 1 for success, or 0 if no matching element is found. Note: The == operation must be defined for the type `elT`.

```
    Int j = -1,kp = -1,kpp = -1;
    Int k = iget(key);
    while (k >= 0) {
        if (j < 0 && el == els[k]) j = k;      Save index of matching el as j.
        kpp = kp;
        kp = k;
        k=nextsis[k];
    }
    if (j < 0) return 0;                       No matching el found.
    if (kpp < 0) {                             The element el was unique under key.
        ierase(key);
        nextsis[j] = -2;
    } else {                                   Patch the list.
        if (j != kp) els[j] = els[kp];         Overwrite j with the terminal element
        nextsis[kpp] = -1;                         and then shorten the list.
        irelinquish(kp);
        nextsis[kp] = -2;
    }
    return 1;                                  Success.
}
```

```
template<class keyT, class elT, class hfnT>
Int Mhash<keyT,elT,hfnT>::count(const keyT &key) {
```

Return the number of elements stored under `key`, 0 if none.

```
    Int next, n = 1;
    if ((next = iget(key)) < 0) return 0;
    while ((next = nextsis[next]) >= 0)  {n++;}
    return n;
}
```

```
template<class keyT, class elT, class hfnT>
Int Mhash<keyT,elT,hfnT>::getinit(const keyT &key) {
```

Initialize `nextget` so that it points to the first element stored under `key`. Return 1 for success, or 0 if no such element.

```
    nextget = iget(key);
    return ((nextget < 0)? 0 : 1);
}
```

```
template<class keyT, class elT, class hfnT>
Int Mhash<keyT,elT,hfnT>::getnext(elT &el) {
```

If `nextget` points validly, copy its element into `el`, update `nextget` to the next element with the same key, and return 1. Otherwise, do not modify `el`, and return 0.

```
    if (nextget < 0) {return 0;}
    el = els[nextget];
    nextget = nextsis[nextget];
    return 1;
}
```

The methods `getinit` and `getnext` are designed to be used in code like this, where `myhash` is a variable of type `Mhash`:

```
Retrieve all elements el stored under a single key and do something with them.
if (myhash.getinit(&key)) {
    while (myhash.getnext(el)) {
        Here use the returned element el.
    }
}
```

7.6.5 Usage Examples

Having exposed in such detail the inner workings of the `Hash` and `Mhash` classes, we may have left the impression that these are difficult to use. Quite the contrary. Here's a piece code that declares a hash memory for integers, and then stores the birth years of some personages:

```
Hash<string,Int,Hashfn2> year(1000,1000);

year[string("Marie Antoinette")] = 1755;
year[string("Ludwig van Beethoven")] = 1770;
year[string("Charles Babbage")] = 1791;
```

As declared, `year` can hold up to 1000 entries. We use the C++ `string` class as the key type. If we want to know how old Marie was when Charles was born, we can write,

```
Int diff = year[string("Charles Babbage")] - year[string("Marie Antoinette")];
cout << diff << '\n';
```

which prints "36".

Instead of using the C++ `string` class, you can, if you must, use null terminated C strings as keys, like this:

```
Hash<char,Int,Hashfn2> yearc(1000,1000);
yearc["Charles Babbage"[0]] = 1791;
```

This works because `Hashfn2` has a special tweak, mentioned above, for key types that are apparently one byte long. Note the required use of `[0]` to send only the first byte of the C string; but that byte is passed by address, so `Hashfn2` knows where to find the rest of the string. (The syntax `yearc[*"Charles Babbage"]` is equivalent, also sending the first byte.)

Suppose we want to go the other direction, namely store the names of people into a hash memory indexed by birth year. Since more than one person may be born in a single year, we want to use a hash multimap memory, `Mhash`:

```
Mhash<Int,string,Hashfn2> person(1000,1000);

person.store(1775, string("Jane Austen"));
person.store(1791, string("Charles Babbage"));
person.store(1767, string("Andrew Jackson"));
person.store(1791, string("James Buchanan"));
person.store(1767, string("John Quincy Adams"));
person.store(1770, string("Ludwig van Beethoven"));
person.store(1791, string("Samuel Morse"));
person.store(1755, string("Marie Antoinette"));
```

It doesn't matter, of course, the order in which we put the names into the hash. Here is a piece of code to loop over years, printing the people born in that year:

```
string str;
for (Int i=1750;i<1800;i++) {
    if (person.getinit(i)) {
        cout << '\n' << "born in " << i << ":\n";
        while (person.getnext(str)) cout << str.data() << '\n';
    }
}
```

which gives as output

```
born in 1755:
Marie Antoinette

born in 1767:
Andrew Jackson
John Quincy Adams

born in 1770:
Ludwig van Beethoven

born in 1775:
Jane Austen

born in 1791:
Charles Babbage
James Buchanan
Samuel Morse
```

Notice that we could *not* have used null terminated C strings in this example, because C++ does not regard them as *first-class objects* that can be stored as elements of a vector. When you are using `Hash` or `Mhash` with strings, you will usually be better off using the C++ `string` class.

In §21.2 and §21.8 we will make extensive use of both the `Hash` and `Mhash` classes and almost all their member functions; look there for further usage examples.

By the way, Mr. Hacher's name is from the French *hacher*, meaning "to mince or hash."

CITED REFERENCES AND FURTHER READING:

Stroustrup, B. 1997, *The C++ Programming Language*, 3rd ed. (Reading, MA: Addison-Wesley), §17.6.2.[1]

Knuth, D.E. 1997, *Sorting and Searching*, 3rd ed., vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §6.4–§6.5.

Vitter, J.S., and Chen, W-C. 1987, *Design and Analysis of Coalesced Hashing* (New York: Oxford University Press).

7.7 Simple Monte Carlo Integration

Inspirations for numerical methods can spring from unlikely sources. "Splines" first were flexible strips of wood used by draftsmen. "Simulated annealing" (we shall

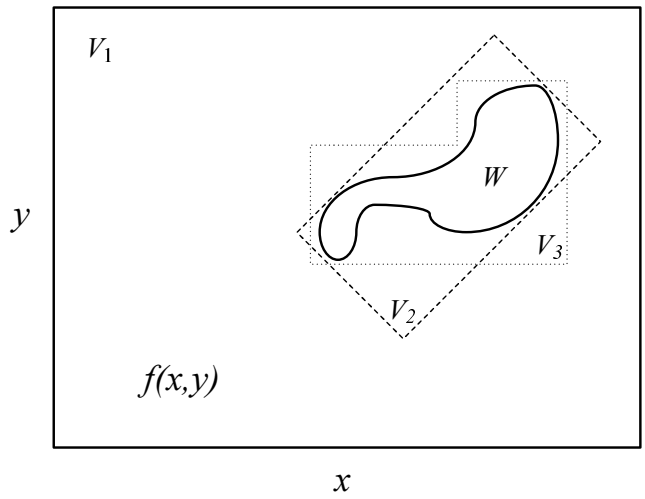

Figure 7.7.1. Monte Carlo integration of a function $f(x, y)$ in a region W. Random points are chosen within an area V that includes W and that can easily be sampled uniformly. Of the three possible V's shown, V_1 is a poor choice because W occupies only a small fraction of its area, while V_2 and V_3 are better choices.

see in §10.12) is rooted in a thermodynamic analogy. And who does not feel at least a faint echo of glamor in the name "Monte Carlo method"?

Suppose that we pick N random points, uniformly distributed in a multidimensional volume V. Call them $x_0, \ldots, x_{N-1}$. Then the basic theorem of Monte Carlo integration estimates the integral of a function f over the multidimensional volume,

$$\int f \, dV \approx V \langle f \rangle \pm V \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}} \tag{7.7.1}$$

Here the angle brackets denote taking the arithmetic mean over the N sample points,

$$\langle f \rangle \equiv \frac{1}{N} \sum_{i=0}^{N-1} f(x_i) \qquad \langle f^2 \rangle \equiv \frac{1}{N} \sum_{i=0}^{N-1} f^2(x_i) \tag{7.7.2}$$

The "plus-or-minus" term in (7.7.1) is a one standard deviation error estimate for the integral, not a rigorous bound; further, there is no guarantee that the error is distributed as a Gaussian, so the error term should be taken only as a rough indication of probable error.

Suppose that you want to integrate a function g over a region W that is not easy to sample randomly. For example, W might have a very complicated shape. No problem. Just find a region V that *includes* W and that *can* easily be sampled, and then define f to be equal to g for points in W and equal to zero for points outside of W (but still inside the sampled V). You want to try to make V enclose W as closely as possible, because the zero values of f will increase the error estimate term of (7.7.1). And well they should: Points chosen outside of W have no information content, so the effective value of N, the number of points, is reduced. The error estimate in (7.7.1) takes this into account.

Figure 7.7.1 shows three possible regions V that might be used to sample a complicated region W. The first, V_1, is obviously a poor choice. A good choice, V_2,

can be sampled by picking a pair of uniform deviates (s, t) and then mapping them into (x, y) by a linear transformation. Another good choice, V_3, can be sampled by, first, using a uniform deviate to choose between the left and right rectangular subregions (in proportion to their respective areas!) and, then, using two more deviates to pick a point inside the chosen rectangle.

Let's create an object that embodies the general scheme described. (We will discuss the implementing code later.) The general idea is to create an `MCintegrate` object by providing (as constructor arguments) the following items:

- a vector `xlo` of lower limits of the coordinates for the rectangular box to be sampled
- a vector `xhi` of upper limits of the coordinates for the rectangular box to be sampled
- a vector-valued function `funcs` that returns as its components one or more functions that we want to integrate simultaneously
- a boolean function that returns whether a point is in the (possibly complicated) region W that we want to integrate; the point will already be within the region V defined by `xlo` and `xhi`
- a mapping function to be discussed below, or `NULL` if there is no mapping function or if your attention span is too short
- a seed for the random number generator

The object `MCintegrate` has this structure.

mcintegrate.h

```
struct MCintegrate {
Object for Monte Carlo integration of one or more functions in an ndim-dimensional region.
    Int ndim,nfun,n;                Number of dimensions, functions, and points sampled.
    VecDoub ff,fferr;               Answers: The integrals and their standard errors.
    VecDoub xlo,xhi,x,xx,fn,sf,sferr;
    Doub vol;                       Volume of the box V.

    VecDoub (*funcsp)(const VecDoub &);      Pointers to the user-supplied functions.
    VecDoub (*xmapp)(const VecDoub &);
    Bool (*inregionp)(const VecDoub &);
    Ran ran;                                 Random number generator.

    MCintegrate(const VecDoub &xlow, const VecDoub &xhigh,
    VecDoub funcs(const VecDoub &), Bool inregion(const VecDoub &),
    VecDoub xmap(const VecDoub &), Int ranseed);
    Constructor. The arguments are in the order described in the itemized list above.

    void step(Int nstep);
    Sample an additional nstep points, accumulating the various sums.

    void calcanswers();
    Calculate answers ff and fferr using the current sums.
};
```

The member function `step` adds sample points, the number of which is given by its argument. The member function `calcanswers` updates the vectors `ff` and `fferr`, which contain respectively the estimated Monte Carlo integrals of the functions and the errors on these estimates. You can examine these values, and then, if you want, call `step` and `calcanswers` again to further reduce the errors.

A worked example will show the underlying simplicity of the method. Suppose that we want to find the weight and the position of the center of mass of an object of

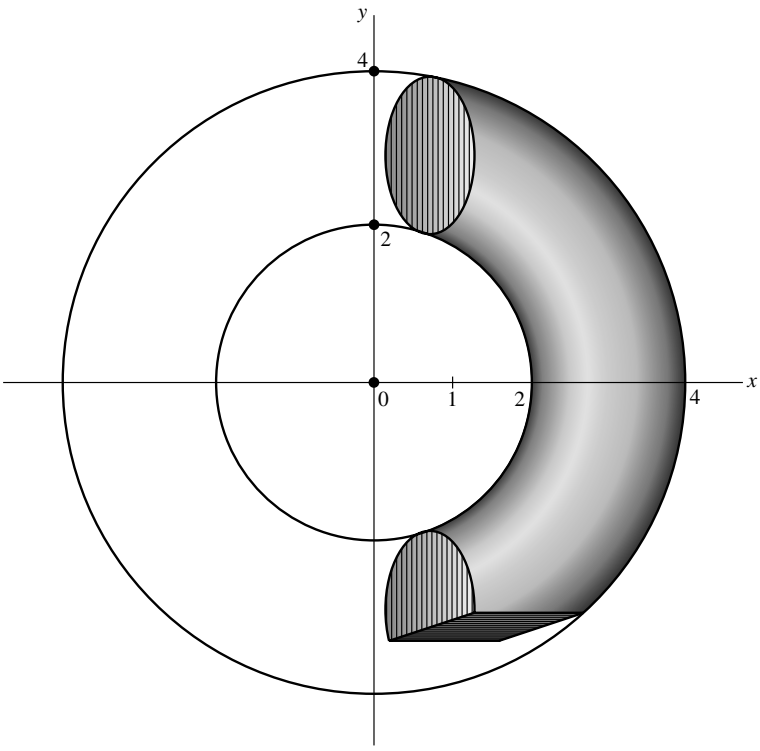

Figure 7.7.2. Example of Monte Carlo integration (see text). The region of interest is a piece of a torus, bounded by the intersection of two planes. The limits of integration of the region cannot easily be written in analytically closed form, so Monte Carlo is a useful technique.

complicated shape, namely the intersection of a torus with the faces of a large box. In particular, let the object be defined by the three simultaneous conditions:

$$z^2 + \left(\sqrt{x^2 + y^2} - 3\right)^2 \leq 1 \tag{7.7.3}$$

(torus centered on the origin with major radius $= 3$, minor radius $= 1$)

$$x \geq 1 \qquad y \geq -3 \tag{7.7.4}$$

(two faces of the box; see Figure 7.7.2). Suppose for the moment that the object has a constant density $\rho = 1$.

We want to estimate the following integrals over the interior of the complicated object:

$$\int \rho \, dx \, dy \, dz \qquad \int x\rho \, dx \, dy \, dz \qquad \int y\rho \, dx \, dy \, dz \qquad \int z\rho \, dx \, dy \, dz \tag{7.7.5}$$

The coordinates of the center of mass will be the ratio of the latter three integrals (linear moments) to the first one (the weight).

To use the `MCintegrate` object, we first write functions that describe the integrands and the region of integration W inside the box V.

mcintegrate.h

```
VecDoub torusfuncs(const VecDoub &x) {
Return the integrands in equation (7.7.5), with ρ = 1.
    Doub den = 1.;
    VecDoub f(4);
    f[0] = den;
    for (Int i=1;i<4;i++) f[i] = x[i-1]*den;
    return f;
}

Bool torusregion(const VecDoub &x) {
Return the inequality (7.7.3).
    return SQR(x[2])+SQR(sqrt(SQR(x[0])+SQR(x[1]))-3.) <= 1.;
}
```

The code to actually do the integration is now quite simple,

```
VecDoub xlo(3), xhi(3);
xlo[0] = 1.; xhi[0] = 4.;
xlo[1] = -3.; xhi[1] = 4.;
xlo[2] = -1.; xhi[2] = 1.;
MCintegrate mymc(xlo,xhi,torusfuncs,torusregion,NULL,10201);
mymc.step(1000000);
mymc.calcanswers();
```

Here we've specified the box V by `xlo` and `xhi`, created an instance of `MCintegrate`, sampled a million times, and updated the answers `mymc.ff` and `mymc.fferr`, which can be accessed for printing or another use.

7.7.1 Change of Variables

A change of variable can often be extremely worthwhile in Monte Carlo integration. Suppose, for example, that we want to evaluate the same integrals, but for a piece of torus whose density is a strong function of z, in fact varying according to

$$\rho(x, y, z) = e^{5z} \tag{7.7.6}$$

One way to do this is, in `torusfuncs`, simply to replace the statement

```
Doub den = 1.;
```

by the statement

```
Doub den = exp(5.*x[2]);
```

This will work, but it is not the best way to proceed. Since (7.7.6) falls so rapidly to zero as z decreases (down to its lower limit -1), most sampled points contribute almost nothing to the sum of the weight or moments. These points are effectively wasted, almost as badly as those that fall outside of the region W. A change of variable, exactly as in the transformation methods of §7.3, solves this problem. Let

$$ds = e^{5z}\,dz \qquad \text{so that} \qquad s = \tfrac{1}{5}e^{5z}, \quad z = \tfrac{1}{5}\ln(5s) \tag{7.7.7}$$

Then $\rho dz = ds$, and the limits $-1 < z < 1$ become $.00135 < s < 29.682$.

The `MCintegrate` object knows that you might want to do this. If it sees an argument `xmap` that is not `NULL`, it will assume that the sampling region defined by `xlo` and `xhi` is not in physical space, but rather needs to be mapped into physical space before either the functions or the region boundary are calculated. Thus, to effect our change of variable, we *don't* need to modify `torusfuncs` or `torusregion`, but we *do* need to modify `xlo` and `xhi`, as well as supply the following function for the argument `xmap`:

mcintegrate.h
```
VecDoub torusmap(const VecDoub &s) {
```
Return the mapping from s to z defined by the last equation in (7.7.7), mapping the other coordinates by the identity map.
```
    VecDoub xx(s);
    xx[2] = 0.2*log(5.*s[2]);
    return xx;
}
```

Code for the actual integration now looks like this:

```
VecDoub slo(3), shi(3);
slo[0] = 1.; shi[0] = 4.;
slo[1] = -3.; shi[1] = 4.;
slo[2] = 0.2*exp(5.*(-1.)); shi[2] = 0.2*exp(5.*(1.));
MCintegrate mymc2(slo,shi,torusfuncs,torusregion,torusmap,10201);
mymc2.step(1000000);
mymc2.calcanswers();
```

If you think for a minute, you will realize that equation (7.7.7) was useful only because the part of the integrand that we wanted to eliminate (e^{5z}) was both integrable analytically and had an integral that could be analytically inverted. (Compare §7.3.2.) In general these properties will not hold. Question: What then? Answer: Pull out of the integrand the "best" factor that *can* be integrated and inverted. The criterion for "best" is to try to reduce the remaining integrand to a function that is as close as possible to constant.

The limiting case is instructive: If you manage to make the integrand f *exactly* constant, and if the region V, of known volume, *exactly* encloses the desired region W, then the average of f that you compute will be exactly its constant value, and the error estimate in equation (7.7.1) will exactly vanish. You will, in fact, have done the integral exactly, and the Monte Carlo numerical evaluations are superfluous. So, backing off from the extreme limiting case, *to the extent* that you are able to make f approximately constant by change of variable, and *to the extent* that you can sample a region only slightly larger than W, you will increase the accuracy of the Monte Carlo integral. This technique is generically called *reduction of variance* in the literature.

The fundamental disadvantage of simple Monte Carlo integration is that its accuracy increases only as the square root of N, the number of sampled points. If your accuracy requirements are modest, or if your computer is large, then the technique is highly recommended as one of great generality. In §7.8 and §7.9 we will see that there are techniques available for "breaking the square root of N barrier" and achieving, at least in some cases, higher accuracy with fewer function evaluations.

There should be nothing surprising in the implementation of `MCintegrate`. The constructor stores pointers to the user functions, makes an otherwise superfluous call to `funcs` just to find out the size of returned vector, and then sizes the sum and answer vectors accordingly. The `step` and `calcanswer` methods implement exactly equations (7.7.1) and (7.7.2).

mcintegrate.h
```
MCintegrate::MCintegrate(const VecDoub &xlow, const VecDoub &xhigh,
    VecDoub funcs(const VecDoub &), Bool inregion(const VecDoub &),
    VecDoub xmap(const VecDoub &), Int ranseed)
    : ndim(xlow.size()), n(0), xlo(xlow), xhi(xhigh), x(ndim), xx(ndim),
    funcsp(funcs), xmapp(xmap), inregionp(inregion), vol(1.), ran(ranseed) {
    if (xmapp) nfun = funcs(xmapp(xlo)).size();
    else nfun = funcs(xlo).size();
    ff.resize(nfun);
    fferr.resize(nfun);
```

```
    fn.resize(nfun);
    sf.assign(nfun,0.);
    sferr.assign(nfun,0.);
    for (Int j=0;j<ndim;j++) vol *= abs(xhi[j]-xlo[j]);
}

void MCintegrate::step(Int nstep) {
    Int i,j;
    for (i=0;i<nstep;i++) {
        for (j=0;j<ndim;j++)
            x[j] = xlo[j]+(xhi[j]-xlo[j])*ran.doub();
        if (xmapp) xx = (*xmapp)(x);
        else xx = x;
        if ((*inregionp)(xx)) {
            fn = (*funcsp)(xx);
            for (j=0;j<nfun;j++) {
                sf[j] += fn[j];
                sferr[j] += SQR(fn[j]);
            }
        }
    }
    n += nstep;
}

void MCintegrate::calcanswers(){
    for (Int j=0;j<nfun;j++) {
        ff[j] = vol*sf[j]/n;
        fferr[j] = vol*sqrt((sferr[j]/n-SQR(sf[j]/n))/n);
    }
}
```

CITED REFERENCES AND FURTHER READING:

Robert, C.P., and Casella, G. 2006, *Monte Carlo Statistical Methods*, 2nd ed. (New York: Springer)

Sobol', I.M. 1994, *A Primer for the Monte Carlo Method* (Boca Raton, FL: CRC Press).

Hammersley, J.M., and Handscomb, D.C. 1964, *Monte Carlo Methods* (London: Methuen).

Gentle, J.E. 2003, *Random Number Generation and Monte Carlo Methods*, 2nd ed. (New York: Springer), Chapter 7.

Shreider, Yu. A. (ed.) 1966, *The Monte Carlo Method* (Oxford: Pergamon).

Kalos, M.H., and Whitlock, P.A. 1986, *Monte Carlo Methods: Volume 1: Basics* (New York: Wiley).

7.8 Quasi- (that is, Sub-) Random Sequences

We have just seen that choosing N points uniformly randomly in an n-dimensional space leads to an error term in Monte Carlo integration that decreases as $1/\sqrt{N}$. In essence, each new point sampled adds linearly to an accumulated sum that will become the function average, and also linearly to an accumulated sum of squares that will become the variance (equation 7.7.2). The estimated error comes from the square root of this variance, hence the power $N^{-1/2}$.

Just because this square-root convergence is familiar does not, however, mean that it is inevitable. A simple counterexample is to choose sample points that lie on a Cartesian grid, and to sample each grid point exactly once (in whatever order).

The Monte Carlo method thus becomes a deterministic quadrature scheme — albeit a simple one — whose fractional error decreases at least as fast as N^{-1} (even faster if the function goes to zero smoothly at the boundaries of the sampled region or is periodic in the region).

The trouble with a grid is that one has to decide *in advance* how fine it should be. One is then committed to completing all of its sample points. With a grid, it is not convenient to "sample *until*" some convergence or termination criterion is met. One might ask if there is not some intermediate scheme, some way to pick sample points "at random," yet spread out in some self-avoiding way, avoiding the chance clustering that occurs with uniformly random points.

A similar question arises for tasks other than Monte Carlo integration. We might want to search an n-dimensional space for a point where some (locally computable) condition holds. Of course, for the task to be computationally meaningful, there had better be continuity, so that the desired condition will hold in some finite n-dimensional neighborhood. We may not know a priori how large that neighborhood is, however. We want to "sample *until*" the desired point is found, moving smoothly to finer scales with increasing samples. Is there any way to do this that is better than uncorrelated, random samples?

The answer to the above question is "yes." Sequences of n-tuples that fill n-space more uniformly than uncorrelated random points are called *quasi-random sequences*. That term is somewhat of a misnomer, since there is nothing "random" about quasi-random sequences: They are cleverly crafted to be, in fact, *sub*random. The sample points in a quasi-random sequence are, in a precise sense, "maximally avoiding" of each other.

A conceptually simple example is *Halton's sequence* [1]. In one dimension, the jth number H_j in the sequence is obtained by the following steps: (i) Write j as a number in base b, where b is some prime. (For example, $j = 17$ in base $b = 3$ is 122.) (ii) Reverse the digits and put a radix point (i.e., a decimal point base b) in front of the sequence. (In the example, we get 0.221 base 3.) The result is H_j. To get a sequence of n-tuples in n-space, you make each component a Halton sequence with a different prime base b. Typically, the first n primes are used.

It is not hard to see how Halton's sequence works: Every time the number of digits in j increases by one place, j's digit-reversed fraction becomes a factor of b finer-meshed. Thus the process is one of filling in all the points on a sequence of finer and finer Cartesian grids — and in a kind of maximally spread-out order on each grid (since, e.g., the most rapidly changing digit in j controls the *most* significant digit of the fraction).

Other ways of generating quasi-random sequences have been proposed by Faure, Sobol', Niederreiter, and others. Bratley and Fox [2] provide a good review and references, and discuss a particularly efficient variant of the Sobol' [3] sequence suggested by Antonov and Saleev [4]. It is this Antonov-Saleev variant whose implementation we now discuss.

The Sobol' sequence generates numbers between zero and one directly as binary fractions of length w bits, from a set of w special binary fractions, V_i, $i = 1, 2, \ldots, w$, called *direction numbers*. In Sobol's original method, the jth number X_j is generated by XORing (bitwise exclusive or) together the set of V_i's satisfying the criterion on i, "the ith bit of j is nonzero." As j increments, in other words, different ones of the V_i's flash in and out of X_j on different time scales. V_1 alternates between being present and absent most quickly, while

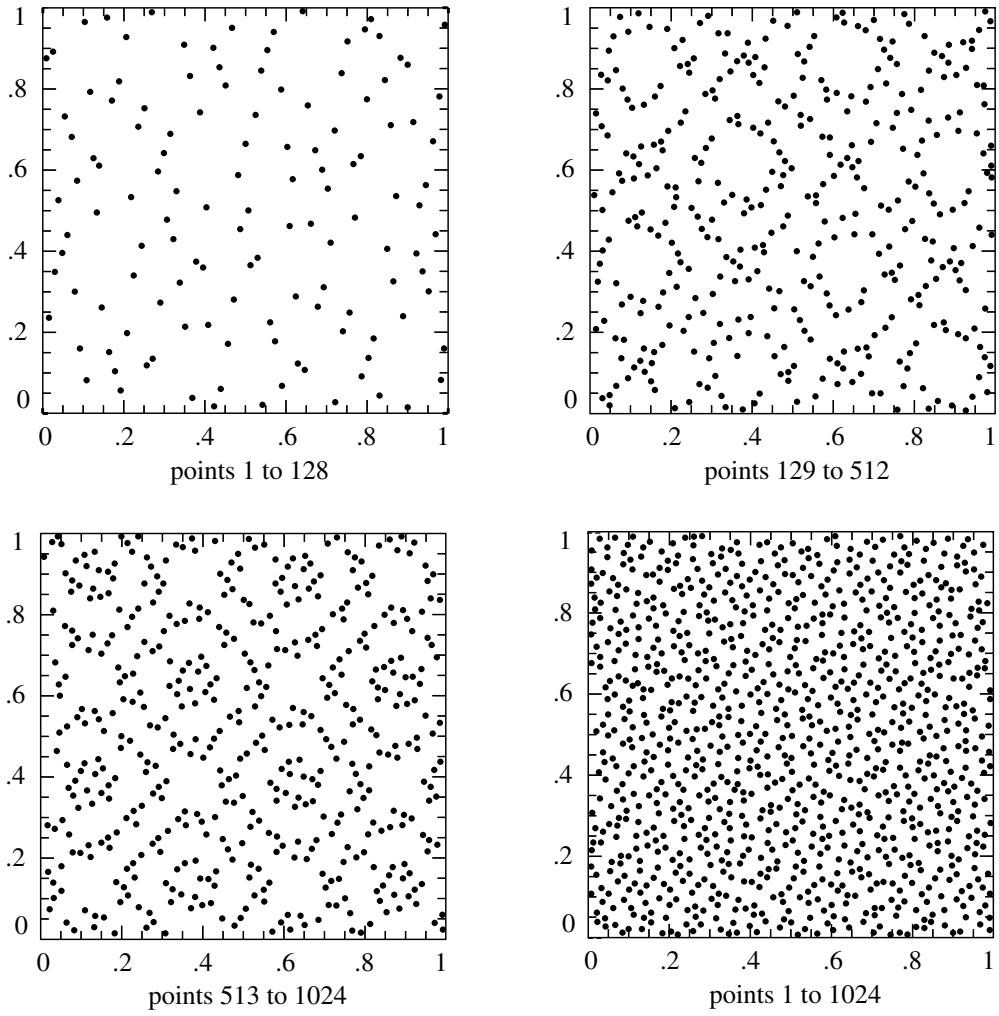

Figure 7.8.1. First 1024 points of a two-dimensional Sobol' sequence. The sequence is generated number-theoretically, rather than randomly, so successive points at any stage "know" how to fill in the gaps in the previously generated distribution.

V_k goes from present to absent (or vice versa) only every 2^{k-1} steps.

Antonov and Saleev's contribution was to show that instead of using the bits of the integer j to select direction numbers, one could just as well use the bits of the *Gray code* of j, $G(j)$. (For a quick review of Gray codes, look at §22.3.)

Now $G(j)$ and $G(j+1)$ differ in exactly one bit position, namely in the position of the rightmost zero bit in the binary representation of j (adding a leading zero to j if necessary). A consequence is that the $j+1$st Sobol'-Antonov-Saleev number can be obtained from the jth by XORing it with *a single* V_i, namely with i the position of the rightmost zero bit in j. This makes the calculation of the sequence very efficient, as we shall see.

Figure 7.8.1 plots the first 1024 points generated by a two-dimensional Sobol' sequence. One sees that successive points do "know" about the gaps left previously, and keep filling them in, hierarchically.

We have deferred to this point a discussion of how the direction numbers V_i are generated. Some nontrivial mathematics is involved in that, so we will content ourselves with a cookbook summary only: Each different Sobol' sequence (or component of an n-dimensional

Degree	Primitive Polynomials Modulo 2*
1	0 (i.e., $x + 1$)
2	1 (i.e., $x^2 + x + 1$)
3	1, 2 (i.e., $x^3 + x + 1$ and $x^3 + x^2 + 1$)
4	1, 4 (i.e., $x^4 + x + 1$ and $x^4 + x^3 + 1$)
5	2, 4, 7, 11, 13, 14
6	1, 13, 16, 19, 22, 25
7	1, 4, 7, 8, 14, 19, 21, 28, 31, 32, 37, 41, 42, 50, 55, 56, 59, 62
8	14, 21, 22, 38, 47, 49, 50, 52, 56, 67, 70, 84, 97, 103, 115, 122
9	8, 13, 16, 22, 25, 44, 47, 52, 55, 59, 62, 67, 74, 81, 82, 87, 91, 94, 103, 104, 109, 122, 124, 137, 138, 143, 145, 152, 157, 167, 173, 176, 181, 182, 185, 191, 194, 199, 218, 220, 227, 229, 230, 234, 236, 241, 244, 253
10	4, 13, 19, 22, 50, 55, 64, 69, 98, 107, 115, 121, 127, 134, 140, 145, 152, 158, 161, 171, 181, 194, 199, 203, 208, 227, 242, 251, 253, 265, 266, 274, 283, 289, 295, 301, 316, 319, 324, 346, 352, 361, 367, 382, 395, 398, 400, 412, 419, 422, 426, 428, 433, 446, 454, 457, 472, 493, 505, 508
*Expressed as a decimal integer whose binary representation gives the coefficients, from the highest to lowest power of x. Only the internal terms are represented — the highest-order term and the constant term always have coefficient 1.	

sequence) is based on a different primitive polynomial over the integers modulo 2, that is, a polynomial whose coefficients are either 0 or 1, and which generates a maximal length shift register sequence. (Primitive polynomials modulo 2 were used in §7.5 and are further discussed in §22.4.) Suppose P is such a polynomial, of degree q,

$$P = x^q + a_1 x^{q-1} + a_2 x^{q-2} + \cdots + a_{q-1} x + 1 \tag{7.8.1}$$

Define a sequence of integers M_i by the q-term recurrence relation,

$$M_i = 2a_1 M_{i-1} \oplus 2^2 a_2 M_{i-2} \oplus \cdots \oplus 2^{q-1} M_{i-q+1} a_{q-1} \oplus (2^q M_{i-q} \oplus M_{i-q}) \tag{7.8.2}$$

Here bitwise XOR is denoted by $\oplus$. The starting values for this recurrence are that $M_1, \ldots, M_q$ can be arbitrary odd integers less than $2, \ldots, 2^q$, respectively. Then, the direction numbers V_i are given by

$$V_i = M_i / 2^i \qquad i = 1, \ldots, w \tag{7.8.3}$$

The table above lists all primitive polynomials modulo 2 with degree $q \leq 10$. Since the coefficients are either 0 or 1, and since the coefficients of x^q and of 1 are predictably 1, it is convenient to denote a polynomial by its middle coefficients taken as the bits of a binary number (higher powers of x being more significant bits). The table uses this convention.

Turn now to the implementation of the Sobol' sequence. Successive calls to the function `sobseq` (after a preliminary initializing call) return successive points in an n-dimensional Sobol' sequence based on the first n primitive polynomials in the table. As given, the routine is initialized for maximum n of 6 dimensions, and for a word length w of 30 bits. These parameters can be altered by changing `MAXBIT` ($\equiv w$) and `MAXDIM`, and by adding more initializing data to the arrays `ip` (the primitive polynomials from the table above), `mdeg` (their degrees), and `iv` (the starting values for the recurrence, equation 7.8.2). A second table, on the next page, elucidates the initializing data in the routine.

Initializing Values Used in sobseq					
Degree	Polynomial	Starting Values			
1	0	1	(3)	(5)	(15) . . .
2	1	1	1	(7)	(11) . . .
3	1	1	3	7	(5) . . .
3	2	1	3	3	(15) . . .
4	1	1	1	3	13 . . .
4	4	1	1	5	9 . . .
Parenthesized values are not freely specifiable, but are forced by the required recurrence for this degree.					

`void sobseq(const Int n, VecDoub_O &x)` sobseq.h

When `n` is negative, internally initializes a set of `MAXBIT` direction numbers for each of `MAXDIM` different Sobol' sequences. When `n` is positive (but $\leq$`MAXDIM`), returns as the vector `x[0..n-1]` the next values from `n` of these sequences. (`n` must not be changed between initializations.)

```
{
    const Int MAXBIT=30,MAXDIM=6;
    Int j,k,l;
    Uint i,im,ipp;
    static Int mdeg[MAXDIM]={1,2,3,3,4,4};
    static Uint in;
    static VecUint ix(MAXDIM);
    static NRvector<Uint*> iu(MAXBIT);
    static Uint ip[MAXDIM]={0,1,1,2,1,4};
    static Uint iv[MAXDIM*MAXBIT]=
        {1,1,1,1,1,1,3,1,3,3,1,1,5,7,7,3,3,5,15,11,5,15,13,9};
    static Doub fac;

    if (n < 0) {                                    Initialize, don't return a vector.
        for (k=0;k<MAXDIM;k++) ix[k]=0;
        in=0;
        if (iv[0] != 1) return;
        fac=1.0/(1 << MAXBIT);
        for (j=0,k=0;j<MAXBIT;j++,k+=MAXDIM) iu[j] = &iv[k];
        To allow both 1D and 2D addressing.
        for (k=0;k<MAXDIM;k++) {
            for (j=0;j<mdeg[k];j++) iu[j][k] <<= (MAXBIT-1-j);
            Stored values only require normalization.
            for (j=mdeg[k];j<MAXBIT;j++) {          Use the recurrence to get other val-
                ipp=ip[k];                              ues.
                i=iu[j-mdeg[k]][k];
                i ^= (i >> mdeg[k]);
                for (l=mdeg[k]-1;l>=1;l--) {
                    if (ipp & 1) i ^= iu[j-l][k];
                    ipp >>= 1;
                }
                iu[j][k]=i;
            }
        }
    } else {                                        Calculate the next vector in the se-
        im=in++;                                        quence.
        for (j=0;j<MAXBIT;j++) {                    Find the rightmost zero bit.
            if (!(im & 1)) break;
            im >>= 1;
        }
```

```
            if (j >= MAXBIT) throw("MAXBIT too small in sobseq");
            im=j*MAXDIM;
            for (k=0;k<MIN(n,MAXDIM);k++) {
                ix[k] ^= iv[im+k];
                x[k]=ix[k]*fac;
            }
        }
    }
```

XOR the appropriate direction number into each component of the vector and convert to a floating number.

How good is a Sobol' sequence, anyway? For Monte Carlo integration of a smooth function in n dimensions, the answer is that the fractional error will decrease with N, the number of samples, as $(\ln N)^n/N$, i.e., almost as fast as $1/N$. As an example, let us integrate a function that is nonzero inside a torus (doughnut) in three-dimensional space. If the major radius of the torus is R_0 and the minor radius is r_0, the minor radial coordinate r is defined by

$$r = \left([(x^2 + y^2)^{1/2} - R_0]^2 + z^2\right)^{1/2} \tag{7.8.4}$$

Let us try the function

$$f(x, y, z) = \begin{cases} 1 + \cos\left(\dfrac{\pi r^2}{r_0^2}\right) & r < r_0 \\ 0 & r \geq r_0 \end{cases} \tag{7.8.5}$$

which can be integrated analytically in cylindrical coordinates, giving

$$\iiint dx\, dy\, dz\, f(x, y, z) = 2\pi^2 r_0^2 R_0 \tag{7.8.6}$$

With parameters $R_0 = 0.6$, $r_0 = 0.3$, we did 100 successive Monte Carlo integrations of equation (7.8.4), sampling uniformly in the region $-1 < x, y, z < 1$, for the two cases of uncorrelated random points and the Sobol' sequence generated by the routine `sobseq`. Figure 7.8.2 shows the results, plotting the r.m.s. average error of the 100 integrations as a function of the number of points sampled. (For any *single* integration, the error of course wanders from positive to negative, or vice versa, so a logarithmic plot of fractional error is not very informative.) The thin, dashed curve corresponds to uncorrelated random points and shows the familiar $N^{-1/2}$ asymptotics. The thin, solid gray curve shows the result for the Sobol' sequence. The logarithmic term in the expected $(\ln N)^3/N$ is readily apparent as curvature in the curve, but the asymptotic N^{-1} is unmistakable.

To understand the importance of Figure 7.8.2, suppose that a Monte Carlo integration of f with 1% accuracy is desired. The Sobol' sequence achieves this accuracy in a few thousand samples, while pseudo-random sampling requires nearly 100,000 samples. The ratio would be even greater for higher desired accuracies.

A different, not quite so favorable, case occurs when the function being integrated has hard (discontinuous) boundaries inside the sampling region, for example the function that is one inside the torus and zero outside,

$$f(x, y, z) = \begin{cases} 1 & r < r_0 \\ 0 & r \geq r_0 \end{cases} \tag{7.8.7}$$

where r is defined in equation (7.8.4). Not by coincidence, this function has the same analytic integral as the function of equation (7.8.5), namely $2\pi^2 r_0^2 R_0$.

The carefully hierarchical Sobol' sequence is based on a set of Cartesian grids, but the boundary of the torus has no particular relation to those grids. The result is that it is essentially random whether sampled points in a thin layer at the surface of the torus, containing on the order of $N^{2/3}$ points, come out to be inside or outside the torus. The square root law, applied to this thin layer, gives $N^{1/3}$ fluctuations in the sum, or $N^{-2/3}$ fractional error in the Monte Carlo integral. One sees this behavior verified in Figure 7.8.2 by the thicker gray curve. The

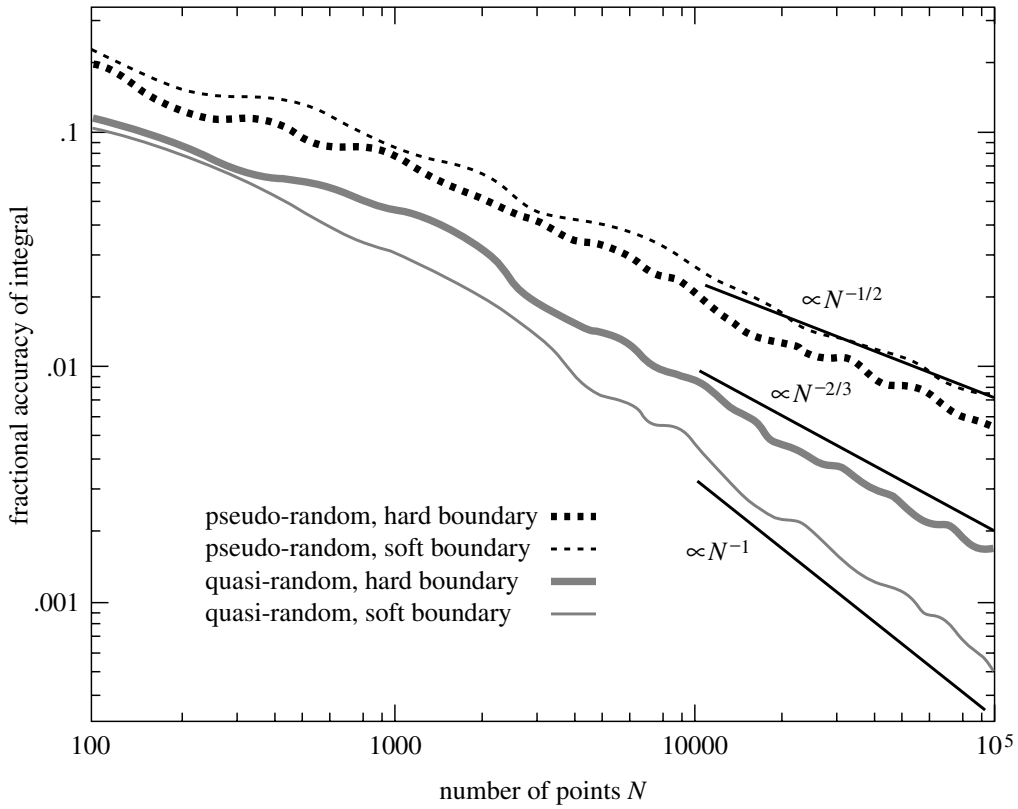

Figure 7.8.2. Fractional accuracy of Monte Carlo integrations as a function of number of points sampled, for two different integrands and two different methods of choosing random points. The quasi-random Sobol' sequence converges much more rapidly than a conventional pseudo-random sequence. Quasi-random sampling does better when the integrand is smooth ("soft boundary") than when it has step discontinuities ("hard boundary"). The curves shown are the r.m.s. averages of 100 trials.

thicker dashed curve in Figure 7.8.2 is the result of integrating the function of equation (7.8.7) using independent random points. While the advantage of the Sobol' sequence is not quite so dramatic as in the case of a smooth function, it can nonetheless be a significant factor ($\sim$5) even at modest accuracies like 1%, and greater at higher accuracies.

Note that we have not provided the routine `sobseq` with a means of starting the sequence at a point other than the beginning, but this feature would be easy to add. Once the initialization of the direction numbers `iv` has been done, the jth point can be obtained directly by XORing together those direction numbers corresponding to nonzero bits in the Gray code of j, as described above.

7.8.1 The Latin Hypercube

We mention here the unrelated technique of *Latin square* or *Latin hypercube* sampling, which is useful when you must sample an N-dimensional space *exceedingly* sparsely, at M points. For example, you may want to test the crashworthiness of cars as a simultaneous function of four different design parameters, but with a budget of only three expendable cars. (The issue is not whether this is a good plan — it isn't — but rather how to make the best of the situation!)

The idea is to partition each design parameter (dimension) into M segments, so that the whole space is partitioned into M^N cells. (You can choose the segments

in each dimension to be equal or unequal, according to taste.) With four parameters and three cars, for example, you end up with $3 \times 3 \times 3 \times 3 = 81$ cells.

Next, choose M cells to contain the sample points by the following algorithm: Randomly choose one of the M^N cells for the first point. Now eliminate all cells that agree with this point on *any* of its parameters (that is, cross out all cells in the same row, column, etc.), leaving $(M-1)^N$ candidates. Randomly choose one of these, eliminate new rows and columns, and continue the process until there is only one cell left, which then contains the final sample point.

The result of this construction is that *each* design parameter will have been tested in *every one* of its subranges. If the response of the system under test is dominated by *one* of the design parameters (the *main effect*), that parameter will be found with this sampling technique. On the other hand, if there are important *interaction effects* among different design parameters, then the Latin hypercube gives no particular advantage. Use with care.

There is a large field in statistics that deals with *design of experiments*. A brief pedagogical introduction is [5].

CITED REFERENCES AND FURTHER READING:

Halton, J.H. 1960, "On the Efficiency of Certain Quasi-Random Sequences of Points in Evaluating Multi-dimensional Integrals," *Numerische Mathematik*, vol. 2, pp. 84–90.[1]

Bratley P., and Fox, B.L. 1988, "Implementing Sobol's Quasirandom Sequence Generator," *ACM Transactions on Mathematical Software*, vol. 14, pp. 88–100.[2]

Lambert, J.P. 1988, "Quasi-Random Sequences in Numerical Practice," in *Numerical Mathematics — Singapore 1988*, ISNM vol. 86, R.P. Agarwal, Y.M. Chow, and S.J. Wilson, eds. (Basel: Birkhäuser), pp. 273–284.

Niederreiter, H. 1988, "Quasi-Monte Carlo Methods for Multidimensional Numerical Integration." in *Numerical Integration III*, ISNM vol. 85, H. Brass and G. Hämmerlin, eds. (Basel: Birkhäuser), pp. 157–171.

Sobol', I.M. 1967, "On the Distribution of Points in a Cube and the Approximate Evaluation of Integrals," *USSR Computational Mathematics and Mathematical Physics*, vol. 7, no. 4, pp. 86–112.[3]

Antonov, I.A., and Saleev, V.M 1979, "An Economic Method of Computing lp_t Sequences," *USSR Computational Mathematics and Mathematical Physics*, vol. 19, no. 1, pp. 252–256.[4]

Dunn, O.J., and Clark, V.A. 1974, *Applied Statistics: Analysis of Variance and Regression* (New York, Wiley) [discusses Latin Square].

Czitrom, V. 1999, "One-Factor-at-a-Time Versus Designed Experiments," *The American Statistician*, vol. 53, pp. 126–131, online at `http://www.amstat.org/publications/tas/czitrom.pdf`.[5]

7.9 Adaptive and Recursive Monte Carlo Methods

This section discusses more advanced techniques of Monte Carlo integration. As examples of the use of these techniques, we include two rather different, fairly sophisticated, multidimensional Monte Carlo codes: `vegas` [1,2], and `miser` [4]. The techniques that we discuss all fall under the general rubric of *reduction of variance* (§7.7), but are otherwise quite distinct.

7.9.1 Importance Sampling

The use of *importance sampling* was already implicit in equations (7.7.6) and (7.7.7). We now return to it in a slightly more formal way. Suppose that an integrand f can be written as the product of a function h that is almost constant times another, positive, function g. Then its integral over a multidimensional volume V is

$$\int f\,dV = \int (f/g)\,g\,dV = \int h\,g\,dV \tag{7.9.1}$$

In equation (7.7.7) we interpreted equation (7.9.1) as suggesting a change of variable to G, the indefinite integral of g. That made $g\,dV$ a perfect differential. We then proceeded to use the basic theorem of Monte Carlo integration, equation (7.7.1). A more general interpretation of equation (7.9.1) is that we can integrate f by instead sampling h — not, however, with uniform probability density dV, but rather with nonuniform density $g\,dV$. In this second interpretation, the first interpretation follows as the special case, where the *means* of generating the nonuniform sampling of $g\,dV$ is via the transformation method, using the indefinite integral G (see §7.3).

More directly, one can go back and generalize the basic theorem (7.7.1) to the case of nonuniform sampling: Suppose that points x_i are chosen within the volume V with a probability density p satisfying

$$\int p\,dV = 1 \tag{7.9.2}$$

The generalized fundamental theorem is that the integral of any function f is estimated, using N sample points $x_0, \ldots, x_{N-1}$, by

$$I \equiv \int f\,dV = \int \frac{f}{p}\,p\,dV \approx \left\langle \frac{f}{p} \right\rangle \pm \sqrt{\frac{\langle f^2/p^2\rangle - \langle f/p\rangle^2}{N}} \tag{7.9.3}$$

where angle brackets denote arithmetic means over the N points, exactly as in equation (7.7.2). As in equation (7.7.1), the "plus-or-minus" term is a one standard deviation error estimate. Notice that equation (7.7.1) is in fact the special case of equation (7.9.3), with $p = \text{constant} = 1/V$.

What is the best choice for the sampling density p? Intuitively, we have already seen that the idea is to make $h = f/p$ as close to constant as possible. We can be more rigorous by focusing on the numerator inside the square root in equation (7.9.3), which is the variance per sample point. Both angle brackets are themselves Monte Carlo estimators of integrals, so we can write

$$S \equiv \left\langle \frac{f^2}{p^2} \right\rangle - \left\langle \frac{f}{p} \right\rangle^2 \approx \int \frac{f^2}{p^2}\,p\,dV - \left[\int \frac{f}{p}\,p\,dV\right]^2 = \int \frac{f^2}{p}\,dV - \left[\int f\,dV\right]^2 \tag{7.9.4}$$

We now find the optimal p subject to the constraint equation (7.9.2) by the functional variation

$$0 = \frac{\delta}{\delta p}\left(\int \frac{f^2}{p}\,dV - \left[\int f\,dV\right]^2 + \lambda \int p\,dV\right) \tag{7.9.5}$$

with λ a Lagrange multiplier. Note that the middle term does not depend on p. The variation (which comes inside the integrals) gives $0 = -f^2/p^2 + \lambda$ or

$$p = \frac{|f|}{\sqrt{\lambda}} = \frac{|f|}{\int |f|\,dV} \tag{7.9.6}$$

where λ has been chosen to enforce the constraint (7.9.2).

If f has one sign in the region of integration, then we get the obvious result that the optimal choice of p — if one can figure out a practical way of effecting the sampling — is that it be proportional to $|f|$. Then the variance is reduced to zero. Not so obvious, but seen

to be true, is the fact that $p \propto |f|$ is optimal even if f takes on both signs. In that case the variance per sample point (from equations 7.9.4 and 7.9.6) is

$$S = S_{\text{optimal}} = \left(\int |f| \, dV \right)^2 - \left(\int f \, dV \right)^2 \tag{7.9.7}$$

One curiosity is that one can add a constant to the integrand to make it all of one sign, since this changes the integral by a known amount, constant$\times V$. Then, the optimal choice of p always gives zero variance, that is, a perfectly accurate integral! The resolution of this seeming paradox (already mentioned at the end of §7.7) is that perfect knowledge of p in equation (7.9.6) requires perfect knowledge of $\int |f| dV$, which is tantamount to already knowing the integral you are trying to compute!

If your function f takes on a known constant value in most of the volume V, it is certainly a good idea to add a constant so as to make that value zero. Having done that, the accuracy attainable by importance sampling depends in practice not on how small equation (7.9.7) is, but rather on how small is equation (7.9.4) for an *implementable* p, likely only a crude approximation to the ideal.

7.9.2 Stratified Sampling

The idea of *stratified sampling* is quite different from importance sampling. Let us expand our notation slightly and let $\langle\!\langle f \rangle\!\rangle$ denote the true average of the function f over the volume V (namely the integral divided by V), while $\langle f \rangle$ denotes as before the simplest (uniformly sampled) Monte Carlo *estimator* of that average:

$$\langle\!\langle f \rangle\!\rangle \equiv \frac{1}{V} \int f \, dV \qquad \langle f \rangle \equiv \frac{1}{N} \sum_i f(x_i) \tag{7.9.8}$$

The variance of the estimator, Var $(\langle f \rangle)$, which measures the square of the error of the Monte Carlo integration, is asymptotically related to the variance of the function, Var $(f) \equiv \langle\!\langle f^2 \rangle\!\rangle - \langle\!\langle f \rangle\!\rangle^2$, by the relation

$$\text{Var}\,(\langle f \rangle) = \frac{\text{Var}\,(f)}{N} \tag{7.9.9}$$

(compare equation 7.7.1).

Suppose we divide the volume V into two equal, disjoint subvolumes, denoted a and b, and sample $N/2$ points in each subvolume. Then another estimator for $\langle\!\langle f \rangle\!\rangle$, different from equation (7.9.8), which we denote $\langle f \rangle'$, is

$$\langle f \rangle' \equiv \tfrac{1}{2} (\langle f \rangle_a + \langle f \rangle_b) \tag{7.9.10}$$

in other words, the mean of the sample averages in the two half-regions. The variance of estimator (7.9.10) is given by

$$\begin{aligned} \text{Var}\left(\langle f \rangle'\right) &= \frac{1}{4} \left[\text{Var}\,(\langle f \rangle_a) + \text{Var}\,(\langle f \rangle_b) \right] \\ &= \frac{1}{4} \left[\frac{\text{Var}_a\,(f)}{N/2} + \frac{\text{Var}_b\,(f)}{N/2} \right] \\ &= \frac{1}{2N} \left[\text{Var}_a\,(f) + \text{Var}_b\,(f) \right] \end{aligned} \tag{7.9.11}$$

Here $\text{Var}_a\,(f)$ denotes the variance of f in subregion a, that is, $\langle\!\langle f^2 \rangle\!\rangle_a - \langle\!\langle f \rangle\!\rangle_a^2$, and correspondingly for b.

From the definitions already given, it is not difficult to prove the relation

$$\text{Var}\,(f) = \tfrac{1}{2} \left[\text{Var}_a\,(f) + \text{Var}_b\,(f) \right] + \tfrac{1}{4} \left(\langle\!\langle f \rangle\!\rangle_a - \langle\!\langle f \rangle\!\rangle_b \right)^2 \tag{7.9.12}$$

(In physics, this formula for combining second moments is the "parallel axis theorem.") Comparing equations (7.9.9), (7.9.11), and (7.9.12), one sees that the stratified (into two subvolumes) sampling gives a variance that is never larger than the simple Monte Carlo case — and smaller whenever the means of the stratified samples, $\langle\langle f \rangle\rangle_a$ and $\langle\langle f \rangle\rangle_b$, are different.

We have not yet exploited the possibility of sampling the two subvolumes with *different numbers* of points, say N_a in subregion a and $N_b \equiv N - N_a$ in subregion b. Let us do so now. Then the variance of the estimator is

$$\mathrm{Var}\left(\langle f \rangle'\right) = \frac{1}{4}\left[\frac{\mathrm{Var}_a(f)}{N_a} + \frac{\mathrm{Var}_b(f)}{N - N_a}\right] \tag{7.9.13}$$

which is minimized (one can easily verify) when

$$\frac{N_a}{N} = \frac{\sigma_a}{\sigma_a + \sigma_b} \tag{7.9.14}$$

Here we have adopted the shorthand notation $\sigma_a \equiv [\mathrm{Var}_a(f)]^{1/2}$, and correspondingly for b. If N_a satisfies equation (7.9.14), then equation (7.9.13) reduces to

$$\mathrm{Var}\left(\langle f \rangle'\right) = \frac{(\sigma_a + \sigma_b)^2}{4N} \tag{7.9.15}$$

Equation (7.9.15) reduces to equation (7.9.9) if $\mathrm{Var}(f) = \mathrm{Var}_a(f) = \mathrm{Var}_b(f)$, in which case stratifying the sample makes no difference.

A standard way to generalize the above result is to consider the volume V divided into more than two equal subregions. One can readily obtain the result that the optimal allocation of sample points among the regions is to have the number of points in each region j proportional to σ_j (that is, the square root of the variance of the function f in that subregion). In spaces of high dimensionality (say $d \gtrsim 4$) this is not in practice very useful, however. Dividing a volume into K segments along each dimension implies K^d subvolumes, typically much too large a number when one contemplates estimating all the corresponding σ_j's.

7.9.3 Mixed Strategies

Importance sampling and stratified sampling seem, at first sight, inconsistent with each other. The former concentrates sample points where the magnitude of the integrand $|f|$ is largest, the latter where the variance of f is largest. How can both be right?

The answer is that (like so much else in life) it all depends on what you know and how well you know it. Importance sampling depends on already knowing some approximation to your integral, so that you are able to generate random points x_i with the desired probability density p. To the extent that your p is not ideal, you are left with an error that decreases only as $N^{-1/2}$. Things are particularly bad if your p is far from ideal in a region where the integrand f is changing rapidly, since then the sampled function $h = f/p$ will have a large variance. Importance sampling works by smoothing the values of the sampled function h and is effective only to the extent that you succeed in this.

Stratified sampling, by contrast, does not necessarily require that you know anything about f. Stratified sampling works by smoothing out the fluctuations of the *number* of points in subregions, not by smoothing the values of the points. The simplest stratified strategy, dividing V into N equal subregions and choosing one point randomly in each subregion, already gives a method whose error decreases asymptotically as N^{-1}, much faster than $N^{-1/2}$. (Note that quasi-random numbers, §7.8, are another way of smoothing fluctuations in the density of points, giving nearly as good a result as the "blind" stratification strategy.)

However, "asymptotically" is an important caveat: For example, if the integrand is negligible in all but a single subregion, then the resulting one-sample integration is all but useless. Information, even very crude, allowing importance sampling to put many points in the active subregion would be much better than blind stratified sampling.

Stratified sampling really comes into its own if you have some way of estimating the variances, so that you can put unequal numbers of points in different subregions, according

to (7.9.14) or its generalizations, *and* if you can find a way of dividing a region into a practical number of subregions (notably *not* K^d with large dimension d), while yet significantly reducing the variance of the function in each subregion compared to its variance in the full volume. Doing this requires a lot of knowledge about f, though different knowledge from what is required for importance sampling.

In practice, importance sampling and stratified sampling are not incompatible. In many, if not most, cases of interest, the integrand f is small everywhere in V except for a small fractional volume of "active regions." In these regions the magnitude of $|f|$ and the standard deviation $\sigma = [\mathrm{Var}(f)]^{1/2}$ are comparable in size, so both techniques will give about the same concentration of points. In more sophisticated implementations, it is also possible to "nest" the two techniques, so that, e.g., importance sampling on a crude grid is followed by stratification within each grid cell.

7.9.4 Adaptive Monte Carlo: VEGAS

The VEGAS algorithm, invented by Peter Lepage [1,2], is widely used for multidimensional integrals that occur in elementary particle physics. VEGAS is primarily based on importance sampling, but it also does some stratified sampling if the dimension d is small enough to avoid K^d explosion (specifically, if $(K/2)^d < N/2$, with N the number of sample points). The basic technique for importance sampling in VEGAS is to construct, adaptively, a multidimensional weight function g that is *separable*,

$$p \propto g(x, y, z, \ldots) = g_x(x) g_y(y) g_z(z) \ldots \tag{7.9.16}$$

Such a function avoids the K^d explosion in two ways: (i) It can be stored in the computer as d separate one-dimensional functions, each defined by K tabulated values, say — so that $K \times d$ replaces K^d. (ii) It can be sampled as a probability density by consecutively sampling the d one-dimensional functions to obtain coordinate vector components $(x, y, z, \ldots)$.

The optimal separable weight function can be shown to be [1]

$$g_x(x) \propto \left[\int dy \int dz \ldots \frac{f^2(x, y, z, \ldots)}{g_y(y) g_z(z) \ldots} \right]^{1/2} \tag{7.9.17}$$

(and correspondingly for $y, z, \ldots$). Notice that this reduces to $g \propto |f|$ (7.9.6) in one dimension. Equation (7.9.17) immediately suggests VEGAS' adaptive strategy: Given a set of g-functions (initially all constant, say), one samples the function f, accumulating not only the overall estimator of the integral, but also the Kd estimators (K subdivisions of the independent variable in each of d dimensions) of the right-hand side of equation (7.9.17). These then determine improved g functions for the next iteration.

When the integrand f is concentrated in one, or at most a few, regions in d-space, then the weight function g's quickly become large at coordinate values that are the projections of these regions onto the coordinate axes. The accuracy of the Monte Carlo integration is then enormously enhanced over what simple Monte Carlo would give.

The weakness of VEGAS is the obvious one: To the extent that the projection of the function f onto individual coordinate directions is uniform, VEGAS gives no concentration of sample points in those dimensions. The worst case for VEGAS, e.g., is an integrand that is concentrated close to a body diagonal line, e.g., one from $(0, 0, 0, \ldots)$ to $(1, 1, 1, \ldots)$. Since this geometry is completely nonseparable, VEGAS can give no advantage at all. More generally, VEGAS may not do well when the integrand is concentrated in one-dimensional (or higher) curved trajectories (or hypersurfaces), unless these happen to be oriented close to the coordinate directions.

The routine `vegas` that follows is essentially Lepage's standard version, minimally modified to conform to our conventions. (We thank Lepage for permission to reproduce the program here.) For consistency with other versions of the VEGAS algorithm in circulation, we have preserved original variable names. The parameter `NDMX` is what we have called K, the maximum number of increments along each axis; `MXDIM` is the maximum value of d; some other parameters are explained in the comments.

The vegas routine performs $m =$ itmx statistically independent evaluations of the desired integral, each with $N =$ ncall function evaluations. While statistically independent, these iterations do assist each other, since each one is used to refine the sampling grid for the next one. The results of all iterations are combined into a single best answer, and its estimated error, by the relations

$$I_{\text{best}} = \sum_{i=0}^{m-1} \frac{I_i}{\sigma_i^2} \Bigg/ \sum_{i=0}^{m-1} \frac{1}{\sigma_i^2} \qquad \sigma_{\text{best}} = \left(\sum_{i=0}^{m-1} \frac{1}{\sigma_i^2} \right)^{-1/2} \tag{7.9.18}$$

Also returned is the quantity

$$\chi^2/m \equiv \frac{1}{m-1} \sum_{i=0}^{m-1} \frac{(I_i - I_{\text{best}})^2}{\sigma_i^2} \tag{7.9.19}$$

If this is significantly larger than 1, then the results of the iterations are statistically inconsistent, and the answers are suspect.

Here is the interface to vegas. (The full code is given in [3].)

```
void vegas(VecDoub_I &regn, Doub fxn(VecDoub_I &, const Doub), const Int init,
    const Int ncall, const Int itmx, const Int nprn, Doub &tgral, Doub &sd,
    Doub &chi2a) {
```

Performs Monte Carlo integration of a user-supplied ndim-dimensional function fxn over a rectangular volume specified by regn[0..2*ndim-1], a vector consisting of ndim "lower left" coordinates of the region followed by ndim "upper right" coordinates. The integration consists of itmx iterations, each with approximately ncall calls to the function. After each iteration the grid is refined; more than 5 or 10 iterations are rarely useful. The input flag init signals whether this call is a new start or a subsequent call for additional iterations (see comments in the code). The input flag nprn (normally 0) controls the amount of diagnostic output. Returned answers are tgral (the best estimate of the integral), sd (its standard deviation), and chi2a (χ^2 per degree of freedom, an indicator of whether consistent results are being obtained). See text for further details.

The input flag init can be used to advantage. One might have a call with init=0, ncall=1000, itmx=5 immediately followed by a call with init=1, ncall=100000, itmx=1. The effect would be to develop a sampling grid over five iterations of a small number of samples, then to do a single high accuracy integration on the optimized grid.

To use vegas for the torus example discussed in §7.7 (the density integrand only, say), the function fxn would be

```
Doub torusfunc(const VecDoub &x, const Doub wgt) {
    Doub den = exp(5.*x[2]);
    if (SQR(x[2])+SQR(sqrt(SQR(x[0])+SQR(x[1]))-3.) <= 1.) return den;
    else return 0.;
}
```

and the main code would be

```
    Doub tgral, sd, chi2a;
    VecDoub regn(6);
    regn[0] = 1.; regn[3] = 4.;
    regn[1] = -3.; regn[4] = 4.;
    regn[2] = -1.; regn[5] = 1.;
    vegas(regn,torusfunc,0,10000,10,0,tgral,sd,chi2a);
    vegas(regn,torusfunc,1,900000,1,0,tgral,sd,chi2a);
```

Note that the user-supplied integrand function, fxn, has an argument wgt in addition to the expected evaluation point x. In most applications you ignore wgt inside the function. Occasionally, however, you may want to integrate some additional function or functions along with the principal function f. The integral of any such function g can be estimated by

$$I_g = \sum_i w_i g(\mathbf{x}) \tag{7.9.20}$$

where the w_i's and $\mathbf{x}$'s are the arguments `wgt` and `x`, respectively. It is straightforward to accumulate this sum inside your function `fxn` and to pass the answer back to your main program via global variables. Of course, $g(\mathbf{x})$ had better resemble the principal function f to some degree, since the sampling will be optimized for f.

The full listing of `vegas` is given in a Webnote [3].

7.9.5 Recursive Stratified Sampling

The problem with stratified sampling, we have seen, is that it may not avoid the K^d explosion inherent in the obvious, Cartesian, tessellation of a d-dimensional volume. A technique called *recursive stratified sampling* [4] attempts to do this by successive bisections of a volume, not along all d dimensions, but rather along only one dimension at a time. The starting points are equations (7.9.10) and (7.9.13), applied to bisections of successively smaller subregions.

Suppose that we have a quota of N evaluations of the function f and want to evaluate $\langle f \rangle'$ in the rectangular parallelepiped region $R = (\mathbf{x}_a, \mathbf{x}_b)$. (We denote such a region by the two coordinate vectors of its diagonally opposite corners.) First, we allocate a fraction p of N toward exploring the variance of f in R: We sample pN function values uniformly in R and accumulate the sums that will give the d different pairs of variances corresponding to the d different coordinate directions along which R can be bisected. In other words, in pN samples, we estimate $\mathrm{Var}\,(f)$ in each of the regions resulting from a possible bisection of R,

$$\begin{aligned} R_{ai} &\equiv (\mathbf{x}_a, \mathbf{x}_b - \tfrac{1}{2}\mathbf{e}_i \cdot (\mathbf{x}_b - \mathbf{x}_a)\mathbf{e}_i) \\ R_{bi} &\equiv (\mathbf{x}_a + \tfrac{1}{2}\mathbf{e}_i \cdot (\mathbf{x}_b - \mathbf{x}_a)\mathbf{e}_i, \mathbf{x}_b) \end{aligned} \tag{7.9.21}$$

Here $\mathbf{e}_i$ is the unit vector in the ith coordinate direction, $i = 1, 2, \ldots, d$.

Second, we inspect the variances to find the most favorable dimension i to bisect. By equation (7.9.15), we could, for example, choose that i for which the sum of the square roots of the variance estimators in regions R_{ai} and R_{bi} is minimized. (Actually, as we will explain, we do something slightly different.)

Third, we allocate the remaining $(1-p)N$ function evaluations between the regions R_{ai} and R_{bi}. If we used equation (7.9.15) to choose i, we should do this allocation according to equation (7.9.14).

We now have two parallelepipeds, each with its own allocation of function evaluations for estimating the mean of f. Our "RSS" algorithm now shows itself to be *recursive*: To evaluate the mean in each region, we go back to the sentence beginning "First,..." in the paragraph above equation (7.9.21). (Of course, when the allocation of points to a region falls below some number, we resort to simple Monte Carlo rather than continue with the recursion.)

Finally, we combine the means and also estimated variances of the two subvolumes using equation (7.9.10) and the first line of equation (7.9.11).

This completes the RSS algorithm in its simplest form. Before we describe some additional tricks under the general rubric of "implementation details," we need to return briefly to equations (7.9.13) – (7.9.15) and derive the equations that we actually use instead of these. The right-hand side of equation (7.9.13) applies the familiar scaling law of equation (7.9.9) twice, once to a and again to b. This would be correct if the estimates $\langle f \rangle_a$ and $\langle f \rangle_b$ were each made by simple Monte Carlo, with uniformly random sample points. However, the two estimates of the mean are in fact made recursively. Thus, there is no reason to expect equation (7.9.9) to hold. Rather, we might substitute for equation (7.9.13) the relation,

$$\mathrm{Var}\left(\langle f \rangle'\right) = \frac{1}{4}\left[\frac{\mathrm{Var}_a\,(f)}{N_a^{\alpha}} + \frac{\mathrm{Var}_b\,(f)}{(N - N_a)^{\alpha}}\right] \tag{7.9.22}$$

where α is an unknown constant ≥ 1 (the case of equality corresponding to simple Monte Carlo). In that case, a short calculation shows that $\mathrm{Var}\left(\langle f \rangle'\right)$ is minimized when

$$\frac{N_a}{N} = \frac{\mathrm{Var}_a\,(f)^{1/(1+\alpha)}}{\mathrm{Var}_a\,(f)^{1/(1+\alpha)} + \mathrm{Var}_b\,(f)^{1/(1+\alpha)}} \tag{7.9.23}$$

and that its minimum value is

$$\mathrm{Var}\left(\langle f\rangle'\right) \propto \left[\mathrm{Var}_a\,(f)^{1/(1+\alpha)} + \mathrm{Var}_b\,(f)^{1/(1+\alpha)}\right]^{1+\alpha} \tag{7.9.24}$$

Equations (7.9.22) – (7.9.24) reduce to equations (7.9.13) – (7.9.15) when $\alpha = 1$. Numerical experiments to find a self-consistent value for α find that $\alpha \approx 2$. That is, when equation (7.9.23) with $\alpha = 2$ is used recursively to allocate sample opportunities, the observed variance of the RSS algorithm goes approximately as N^{-2}, while any other value of α in equation (7.9.23) gives a poorer fall-off. (The sensitivity to α is, however, not very great; it is not known whether $\alpha = 2$ is an analytically justifiable result or only a useful heuristic.)

The principal difference between `miser`'s implementation and the algorithm as described thus far lies in how the variances on the right-hand side of equation (7.9.23) are estimated. We find empirically that it is somewhat more robust to use the square of the difference of maximum and minimum sampled function values, instead of the genuine second moment of the samples. This estimator is of course increasingly biased with increasing sample size; however, equation (7.9.23) uses it only to compare two subvolumes (a and b) having approximately equal numbers of samples. The "max minus min" estimator proves its worth when the preliminary sampling yields only a single point, or a small number of points, in active regions of the integrand. In many realistic cases, these are indicators of nearby regions of even greater importance, and it is useful to let them attract the greater sampling weight that "max minus min" provides.

A second modification embodied in the code is the introduction of a "dithering parameter," `dith`, whose nonzero value causes subvolumes to be divided not exactly down the middle, but rather into fractions $0.5 \pm$`dith`, with the sign of the $\pm$ randomly chosen by a built-in random number routine. Normally `dith` can be set to zero. However, there is a large advantage in taking `dith` to be nonzero if some special symmetry of the integrand puts the active region exactly at the midpoint of the region, or at the center of some power-of-two submultiple of the region. One wants to avoid the extreme case of the active region being evenly divided into 2^d abutting corners of a d-dimensional space. A typical nonzero value of `dith`, on those occasions when it is useful, might be 0.1. Of course, when the dithering parameter is nonzero, we must take the differing sizes of the subvolumes into account; the code does this through the variable `fracl`.

One final feature in the code deserves mention. The RSS algorithm uses a single set of sample points to evaluate equation (7.9.23) in all d directions. At the bottom levels of the recursion, the number of sample points can be quite small. Although rare, it can happen that in one direction all the samples are in one half of the volume; in that case, that direction is ignored as a candidate for bifurcation. Even more rare is the possibility that all of the samples are in one half of the volume in *all* directions. In this case, a random direction is chosen. If this happens too often in your application, then you should increase `MNPT` (see line `if (jb == -1)...` in the code).

Note that `miser`, as given, returns as `ave` an estimate of the average function value $\langle\langle f \rangle\rangle$, not the integral of f over the region. The routine `vegas`, adopting the other convention, returns as `tgral` the integral. The two conventions are of course trivially related, by equation (7.9.8), since the volume V of the rectangular region is known.

The interface to the `miser` routine is this:

```
void miser(Doub func(VecDoub_I &), VecDoub_I &regn, const Int npts,
    const Doub dith, Doub &ave, Doub &var) {
```

Monte Carlo samples a user-supplied `ndim`-dimensional function `func` in a rectangular volume specified by `regn[0..2*ndim-1]`, a vector consisting of `ndim` "lower-left" coordinates of the region followed by `ndim` "upper-right" coordinates. The function is sampled a total of `npts` times, at locations determined by the method of recursive stratified sampling. The mean value of the function in the region is returned as `ave`; an estimate of the statistical uncertainty of `ave` (square of standard deviation) is returned as `var`. The input parameter `dith` should normally be set to zero, but can be set to (e.g.) 0.1 if `func`'s active region falls on the boundary of a power-of-two subdivision of `region`.

Implementing code for the torus problem in §7.7 is

```
Doub torusfunc(const VecDoub &x) {
    Doub den = exp(5.*x[2]);
    if (SQR(x[2])+SQR(sqrt(SQR(x[0])+SQR(x[1]))-3.) <= 1.) return den;
    else return 0.;
}
```

and the `main` code is

```
    Doub ave, var, tgral, sd, vol = 3.*7.*2.;
    regn[0] = 1.; regn[3] = 4.;
    regn[1] = -3.; regn[4] = 4.;
    regn[2] = -1.; regn[5] = 1.;
    miser(torusfunc,regn,1000000,0.,ave,var);
    tgral = ave*vol;
    sd = sqrt(var)*vol;
```

(Actually, `miser` is not particularly well-suited to this problem.)

The complete listing of `miser` is given in a Webnote [5]. The `miser` routine calls the short function `ranpt` to get a random point within a specified d-dimensional region. The version of `ranpt` in the Webnote makes consecutive calls to a uniform random number generator and does the obvious scaling. One can easily modify `ranpt` to generate its points via the quasi-random routine `sobseq` (§7.8). We find that `miser` with `sobseq` can be considerably more accurate than `miser` with uniform random deviates. Since the use of RSS and the use of quasi-random numbers are completely separable, however, we have not made the code given here dependent on `sobseq`. A similar remark might be made regarding importance sampling, which could in principle be combined with RSS. (One could in principle combine `vegas` and `miser`, although the programming would be intricate.)

CITED REFERENCES AND FURTHER READING:

Hammersley, J.M. and Handscomb, D.C. 1964, *Monte Carlo Methods* (London: Methuen).

Kalos, M.H. and Whitlock, P.A. 1986, *Monte Carlo Methods* (New York: Wiley).

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation*, 2nd ed. (New York: Springer).

Lepage, G.P. 1978, "A New Algorithm for Adaptive Multidimensional Integration," *Journal of Computational Physics*, vol. 27, pp. 192–203.[1]

Lepage, G.P. 1980, "VEGAS: An Adaptive Multidimensional Integration Program," Publication CLNS-80/447, Cornell University.[2]

Numerical Recipes Software 2007, "Complete VEGAS Code Listing," *Numerical Recipes Webnote No. 9*, at `http://www.nr.com/webnotes?9` [3]

Press, W.H., and Farrar, G.R. 1990, "Recursive Stratified Sampling for Multidimensional Monte Carlo Integration," *Computers in Physics*, vol. 4, pp. 190–195.[4]

Numerical Recipes Software 2007, "Complete Miser Code Listing," *Numerical Recipes Webnote No. 10*, at `http://www.nr.com/webnotes?10` [5]