Chapter 7. Random Numbers

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. Any program, after all, will produce 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, though imprecise, 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 body of random number generators which 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.

A pragmatic point of view, then, is 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 a certain list of statistical tests, some sensible and some merely enshrined by history, which on the whole will do a very good job of ferreting out any correlations that are 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.

As for references on this subject, the one to turn to first is Knuth [1]. Then try [2]. Only a few of the standard books on numerical methods [3-4] treat topics relating to random numbers.

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), Chapter 3, especially §3.5. [1]

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

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 11. [3]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 10. [4]

7.1 Uniform Deviates

Uniform deviates are just random numbers that lie within a specified range (typically 0 to 1), with any one number in the range 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.

System-Supplied Random Number Generators

Most C implementations have, lurking within, a pair of library routines for initializing, and then generating, "random numbers." In ANSI C, the synopsis is:

```
#include <stdlib.h>
#define RAND_MAX ...

void srand(unsigned seed);
int rand(void);
```

You initialize the random number generator by invoking `srand(seed)` with some arbitrary `seed`. Each initializing value will typically result in a different random sequence, or a least a different starting point in some one enormously long sequence. The *same* initializing value of `seed` will always return the *same* random sequence, however.

You obtain successive random numbers in the sequence by successive calls to `rand()`. That function returns an integer that is typically in the range 0 to the largest representable positive value of type `int` (inclusive). Usually, as in ANSI C, this largest value is available as `RAND_MAX`, but sometimes you have to figure it out for yourself. If you want a random `float` value between 0.0 (inclusive) and 1.0 (exclusive), you get it by an expression like

```
x = rand()/(RAND_MAX+1.0);
```

Now our first, and perhaps most important, lesson in this chapter is: be *very, very* suspicious of a system-supplied `rand()` that resembles the one just described. If all scientific papers whose results are in doubt because of bad `rand()`s were to disappear from library shelves, there would be a gap on each shelf about as big as your fist. System-supplied `rand()`s are almost always *linear congruential generators*, which generate a sequence of integers $I_1, I_2, I_3, \ldots$, each between 0 and $m - 1$ (e.g., `RAND_MAX`) by the recurrence relation

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

Here m is called the *modulus*, and a and c are positive integers called the *multiplier* and the *increment* respectively. The recurrence (7.1.1) will 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.

Although this general framework is powerful enough to provide quite decent random numbers, its implementation in many, if not most, ANSI C libraries is quite flawed; quite a number of implementations are in the category "totally botched." Blame should be apportioned about equally between the ANSI C committee and the implementors. The typical problems are these: First, since the ANSI standard specifies that `rand()` return a value of type `int` — which is only a two-byte quantity on many machines — `RAND_MAX` is often *not* very large. The ANSI C standard requires only that it be at least 32767. This can be disastrous in many circumstances: for a Monte Carlo integration (§7.6 and §7.8), you might well want to evaluate 10^6 different points, but actually be evaluating the same 32767 points 30 times each, not at all the same thing! You should categorically reject any library random number routine with a two-byte returned value.

Second, the ANSI committee's published rationale includes the following mischievous passage: "The committee decided that an implementation should be allowed to provide a `rand` function which generates the best random sequence possible in that implementation, and therefore mandated no standard algorithm. It recognized the value, however, of being able to generate the same pseudo-random sequence in different implementations, and so *it has published an example*. . . . [emphasis added]" The "example" is

```
unsigned long next=1;

int rand(void) /* NOT RECOMMENDED (see text) */
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

void srand(unsigned int seed)
{
    next=seed;
}
```

This corresponds to equation (7.1.1) with $a = 1103515245$, $c = 12345$, and $m = 2^{32}$ (since arithmetic done on `unsigned long` quantities is guaranteed to return the correct low-order bits). These are not particularly good choices for a and c (the period is only 2^{30}), though they are not gross embarrassments by themselves. The real botches occur when implementors, taking the committee's statement above as license, try to "improve" on the published example. For example, one popular 32-bit PC-compatible compiler provides a `long` generator that uses the above congruence, but swaps the high-order and low-order 16 bits of the returned value. Somebody probably thought that this extra flourish added randomness; in fact it ruins the generator. While these kinds of blunders can, of course, be fixed, there remains a fundamental flaw in simple linear congruential generators, which we now discuss.

The linear congruential method has the advantage of being very fast, requiring only a few operations per call, hence its almost universal use. It has the disadvantage that it is not free of sequential correlation on successive calls. 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, a, and c are not very carefully chosen, there will be *many fewer than that.* If m is as bad as 32768, then the number of planes on which triples of points lie in three-dimensional space will be no greater than about the cube root of 32768, or 32. Even if m is close to the machine's largest representable integer, e.g., $\sim 2^{32}$, the number of planes on which triples of points lie in three-dimensional space is usually 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, you might be using a generator whose choices of m, a, and c have been botched. 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 [1]. One of us recalls 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." Figure that out.

Correlation in k-space is not the only weakness of linear congruential generators. Such generators often have their low-order (least significant) bits much less random than their high-order bits. If you want to generate a random integer between 1 and 10, you should always do it using high-order bits, as in

```
j=1+(int) (10.0*rand()/(RAND_MAX+1.0));
```

and never by anything resembling

```
j=1+(rand() % 10);
```

(which uses lower-order bits). Similarly you should never try to take apart a "`rand()`" number into several supposedly random pieces. Instead use separate calls for every piece.

Portable Random Number Generators

Park and Miller [1] have surveyed a large number of random number generators that have been used over the last 30 years or more. Along with a good theoretical review, they present an anecdotal sampling of a number of inadequate generators that have come into widespread use. The historical record is nothing if not appalling.

There is good evidence, both theoretical and empirical, that the simple multiplicative congruential algorithm

$$I_{j+1} = aI_j \pmod{m} \tag{7.1.2}$$

can be as good as any of the more general linear congruential generators that have $c \neq 0$ (equation 7.1.1) — *if* the multiplier a and modulus m are chosen exquisitely carefully. Park and Miller propose a "Minimal Standard" generator based on the choices

$$a = 7^5 = 16807 \qquad m = 2^{31} - 1 = 2147483647 \tag{7.1.3}$$

First proposed by Lewis, Goodman, and Miller in 1969, this generator has in subsequent years passed all new theoretical tests, and (perhaps more importantly) has accumulated a large amount of successful use. Park and Miller do not claim that the generator is "perfect" (we will see below that it is not), but only that it is a good minimal standard against which other generators should be judged.

It is not possible to implement equations (7.1.2) and (7.1.3) directly in a high-level language, since the product of a and $m - 1$ exceeds the maximum value for a 32-bit integer. Assembly language implementation using a 64-bit product register is straightforward, but not portable from machine to machine. A trick due to Schrage [2,3] for multiplying two 32-bit integers modulo a 32-bit constant, without using any intermediates larger than 32 bits (including a sign bit) is therefore extremely interesting: It allows the Minimal Standard generator to be implemented in essentially any programming language on essentially any machine.

Schrage's algorithm is based on an *approximate factorization* of m,

$$m = aq + r, \quad \text{i.e.,} \quad q = [m/a],\ r = m \bmod a \tag{7.1.4}$$

with square brackets denoting integer part. If r is small, specifically $r < q$, and $0 < z < m - 1$, it can be shown that both $a(z \bmod q)$ and $r[z/q]$ lie in the range $0, \ldots, m - 1$, and that

$$az \bmod m = \begin{cases} a(z \bmod q) - r[z/q] & \text{if it is} \geq 0, \\ a(z \bmod q) - r[z/q] + m & \text{otherwise} \end{cases} \tag{7.1.5}$$

The application of Schrage's algorithm to the constants (7.1.3) uses the values $q = 127773$ and $r = 2836$.

Here is an implementation of the Minimal Standard generator:

```
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define MASK 123459876

float ran0(long *idum)
"Minimal" random number generator of Park and Miller. Returns a uniform random deviate
between 0.0 and 1.0. Set or reset idum to any integer value (except the unlikely value MASK)
to initialize the sequence; idum must not be altered between calls for successive deviates in
a sequence.
{
    long k;
    float ans;

    *idum ^= MASK;                      XORing with MASK allows use of zero and other
    k=(*idum)/IQ;                           simple bit patterns for idum.
    *idum=IA*(*idum-k*IQ)-IR*k;         Compute idum=(IA*idum) % IM without over-
    if (*idum < 0) *idum += IM;             flows by Schrage's method.
    ans=AM*(*idum);                     Convert idum to a floating result.
    *idum ^= MASK;                      Unmask before return.
    return ans;
}
```

The period of `ran0` is $2^{31} - 2 \approx 2.1 \times 10^9$. A peculiarity of generators of the form (7.1.2) is that the value 0 must never be allowed as the initial seed — it perpetuates itself — and it never occurs for any nonzero initial seed. Experience has shown that users always manage to call random number generators with the seed `idum=0`. That is why `ran0` performs its exclusive-or with an arbitrary constant both on entry and exit. If you are the first user in history to be proof against human error, you can remove the two lines with the $\wedge$ operation.

Park and Miller discuss two other multipliers a that can be used with the same $m = 2^{31} - 1$. These are $a = 48271$ (with $q = 44488$ and $r = 3399$) and $a = 69621$ (with $q = 30845$ and $r = 23902$). These can be substituted in the routine `ran0` if desired; they may be slightly superior to Lewis *et al.*'s longer-tested values. No values other than these should be used.

The routine `ran0` is a Minimal Standard, satisfactory for the majority of applications, but we do not recommend it as the final word on random number generators. Our reason is precisely the simplicity of the Minimal Standard. It is not hard to think of situations where successive random numbers might be used in a way that accidentally conflicts with the generation algorithm. For example, since successive numbers differ by a multiple of only 1.6×10^4 out of a modulus of more than 2×10^9, very small random numbers will tend to be followed by smaller than average values. One time in 10^6, for example, there will be a value $< 10^{-6}$ returned (as there should be), but this will *always* be followed by a value less than about 0.0168. One can easily think of applications involving rare events where this property would lead to wrong results.

There are other, more subtle, serial correlations present in `ran0`. For example, if successive points (I_i, I_{i+1}) are binned into a two-dimensional plane for $i = 1, 2, \ldots, N$, then the resulting distribution fails the χ^2 test when N is greater than a few $\times 10^7$, much less than the period $m - 2$. Since low-order serial correlations have historically been such a bugaboo, and since there is a very simple way to remove

them, we think that it is prudent to do so.

The following routine, ran1, uses the Minimal Standard for its random value, but it shuffles the output to remove low-order serial correlations. A random deviate derived from the jth value in the sequence, I_j, is output not on the jth call, but rather on a randomized later call, $j + 32$ on average. The shuffling algorithm is due to Bays and Durham as described in Knuth [4], and is illustrated in Figure 7.1.1.

```
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define NTAB 32
#define NDIV (1+(IM-1)/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)

float ran1(long *idum)
"Minimal" random number generator of Park and Miller with Bays-Durham shuffle and added
safeguards. Returns a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint
values). Call with idum a negative integer to initialize; thereafter, do not alter idum between
successive deviates in a sequence. RNMX should approximate the largest floating value that is
less than 1.
{
    int j;
    long k;
    static long iy=0;
    static long iv[NTAB];
    float temp;

    if (*idum <= 0 || !iy) {                Initialize.
        if (-(*idum) < 1) *idum=1;          Be sure to prevent idum = 0.
        else *idum = -(*idum);
        for (j=NTAB+7;j>=0;j--) {           Load the shuffle table (after 8 warm-ups).
            k=(*idum)/IQ;
            *idum=IA*(*idum-k*IQ)-IR*k;
            if (*idum < 0) *idum += IM;
            if (j < NTAB) iv[j] = *idum;
        }
        iy=iv[0];
    }
    k=(*idum)/IQ;                           Start here when not initializing.
    *idum=IA*(*idum-k*IQ)-IR*k;             Compute idum=(IA*idum) % IM without over-
    if (*idum < 0) *idum += IM;                 flows by Schrage's method.
    j=iy/NDIV;                              Will be in the range 0..NTAB-1.
    iy=iv[j];                               Output previously stored value and refill the
    iv[j] = *idum;                              shuffle table.
    if ((temp=AM*iy) > RNMX) return RNMX;   Because users don't expect endpoint values.
    else return temp;
}
```

The routine ran1 passes those statistical tests that ran0 is known to fail. In fact, we do not know of any statistical test that ran1 fails to pass, except when the number of calls starts to become on the order of the period m, say $> 10^8 \approx m/20$.

For situations when even longer random sequences are needed, L'Ecuyer [6] has given a good way of combining two different sequences with different periods so as to obtain a new sequence whose period is the least common multiple of the two periods. The basic idea is simply to add the two sequences, modulo the modulus of

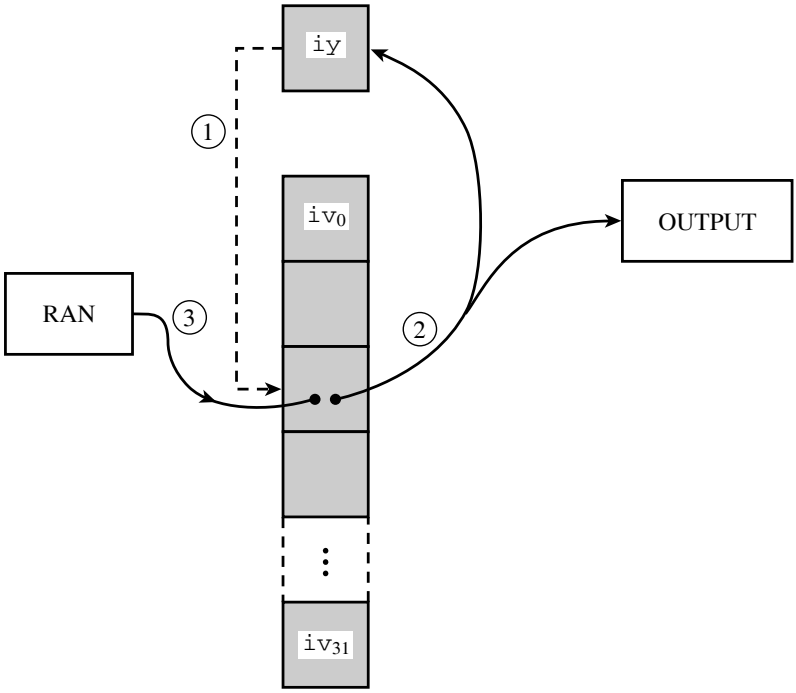

Figure 7.1.1. Shuffling procedure used in `ran1` to break up sequential correlations in the Minimal Standard generator. Circled numbers indicate the sequence of events: On each call, the random number in `iy` is used to choose a random element in the array `iv`. That element becomes the output random number, and also is the next `iy`. Its spot in `iv` is refilled from the Minimal Standard routine.

either of them (call it m). A trick to avoid an intermediate value that overflows the integer wordsize is to subtract rather than add, and then add back the constant $m - 1$ if the result is ≤ 0, so as to wrap around into the desired interval $0, \ldots, m - 1$.

Notice that it is not necessary that this wrapped subtraction be able to reach all values $0, \ldots, m - 1$ from *every* value of the first sequence. Consider the absurd extreme case where the value subtracted was only between 1 and 10: The resulting sequence would still be no less random than the first sequence by itself. As a practical matter it is only necessary that the second sequence have a range covering *substantially* all of the range of the first. L'Ecuyer recommends the use of the two generators $m_1 = 2147483563$ (with $a_1 = 40014$, $q_1 = 53668$, $r_1 = 12211$) and $m_2 = 2147483399$ (with $a_2 = 40692$, $q_2 = 52774$, $r_2 = 3791$). Both moduli are slightly less than 2^{31}. The periods $m_1 - 1 = 2 \times 3 \times 7 \times 631 \times 81031$ and $m_2 - 1 = 2 \times 19 \times 31 \times 1019 \times 1789$ share only the factor 2, so the period of the combined generator is $\approx 2.3 \times 10^{18}$. For present computers, period exhaustion is a practical impossibility.

Combining the two generators breaks up serial correlations to a considerable extent. We nevertheless recommend the additional shuffle that is implemented in the following routine, `ran2`. We think that, within the limits of its floating-point precision, `ran2` provides perfect random numbers; a practical definition of "perfect" is that we will pay $1000 to the first reader who convinces us otherwise (by finding a statistical test that `ran2` fails in a nontrivial way, excluding the ordinary limitations of a machine's floating-point representation).

```
#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)

float ran2(long *idum)
```

Long period ($> 2 \times 10^{18}$) random number generator of L'Ecuyer with Bays-Durham shuffle and added safeguards. Returns a uniform random deviate between 0.0 and 1.0 (exclusive of the endpoint values). Call with `idum` a negative integer to initialize; thereafter, do not alter `idum` between successive deviates in a sequence. `RNMX` should approximate the largest floating value that is less than 1.

```
{
    int j;
    long k;
    static long idum2=123456789;
    static long iy=0;
    static long iv[NTAB];
    float temp;

    if (*idum <= 0) {                       Initialize.
        if (-(*idum) < 1) *idum=1;          Be sure to prevent idum = 0.
        else *idum = -(*idum);
        idum2=(*idum);
        for (j=NTAB+7;j>=0;j--) {           Load the shuffle table (after 8 warm-ups).
            k=(*idum)/IQ1;
            *idum=IA1*(*idum-k*IQ1)-k*IR1;
            if (*idum < 0) *idum += IM1;
            if (j < NTAB) iv[j] = *idum;
        }
        iy=iv[0];
    }
    k=(*idum)/IQ1;                          Start here when not initializing.
    *idum=IA1*(*idum-k*IQ1)-k*IR1;          Compute idum=(IA1*idum) % IM1 without
    if (*idum < 0) *idum += IM1;                overflows by Schrage's method.
    k=idum2/IQ2;
    idum2=IA2*(idum2-k*IQ2)-k*IR2;          Compute idum2=(IA2*idum) % IM2 likewise.
    if (idum2 < 0) idum2 += IM2;
    j=iy/NDIV;                              Will be in the range 0..NTAB-1.
    iy=iv[j]-idum2;                         Here idum is shuffled, idum and idum2 are
    iv[j] = *idum;                              combined to generate output.
    if (iy < 1) iy += IMM1;
    if ((temp=AM*iy) > RNMX) return RNMX;   Because users don't expect endpoint values.
    else return temp;
}
```

L'Ecuyer [6] lists additional short generators that can be combined into longer ones, including generators that can be implemented in 16-bit integer arithmetic.

Finally, we give you Knuth's suggestion [4] for a portable routine, which we have translated to the present conventions as `ran3`. This is not based on the linear congruential method at all, but rather on a *subtractive method* (see also [5]). One might hope that its weaknesses, if any, are therefore of a highly different character

from the weaknesses, if any, of ran1 above. If you ever suspect trouble with one routine, it is a good idea to try the other in the same application. ran3 has one nice feature: if your machine is poor on integer arithmetic (i.e., is limited to 16-bit integers), you can declare mj, mk, and ma[] as float, define mbig and mseed as 4000000 and 1618033, respectively, and the routine will be rendered entirely floating-point.

```
#include <stdlib.h>                          Change to math.h in K&R C.
#define MBIG 1000000000
#define MSEED 161803398
#define MZ 0
#define FAC (1.0/MBIG)
```

According to Knuth, any large MBIG, and any smaller (but still large) MSEED can be substituted for the above values.

```
float ran3(long *idum)
```

Returns a uniform random deviate between 0.0 and 1.0. Set idum to any negative value to initialize or reinitialize the sequence.

```
{
    static int inext,inextp;
    static long ma[56];                      The value 56 (range ma[1..55]) is special and
    static int iff=0;                            should not be modified; see Knuth.
    long mj,mk;
    int i,ii,k;

    if (*idum < 0 || iff == 0) {             Initialization.
        iff=1;
        mj=labs(MSEED-labs(*idum));          Initialize ma[55] using the seed idum and the
        mj %= MBIG;                              large number MSEED.
        ma[55]=mj;
        mk=1;
        for (i=1;i<=54;i++) {                Now initialize the rest of the table,
            ii=(21*i) % 55;                  in a slightly random order,
            ma[ii]=mk;                       with numbers that are not especially random.
            mk=mj-mk;
            if (mk < MZ) mk += MBIG;
            mj=ma[ii];
        }
        for (k=1;k<=4;k++)                   We randomize them by "warming up the gener-
            for (i=1;i<=55;i++) {                ator."
                ma[i] -= ma[1+(i+30) % 55];
                if (ma[i] < MZ) ma[i] += MBIG;
            }
        inext=0;                             Prepare indices for our first generated number.
        inextp=31;                           The constant 31 is special; see Knuth.
        *idum=1;
    }
    Here is where we start, except on initialization.
    if (++inext == 56) inext=1;              Increment inext and inextp, wrapping around
    if (++inextp == 56) inextp=1;                56 to 1.
    mj=ma[inext]-ma[inextp];                 Generate a new random number subtractively.
    if (mj < MZ) mj += MBIG;                 Be sure that it is in range.
    ma[inext]=mj;                            Store it,
    return mj*FAC;                           and output the derived uniform deviate.
}
```

Quick and Dirty Generators

One sometimes would like a "quick and dirty" generator to embed in a program, perhaps taking only one or two lines of code, just to *somewhat* randomize things. One might wish to

process data from an experiment not always in exactly the same order, for example, so that the first output is more "typical" than might otherwise be the case.

For this kind of application, all we really need is a list of "good" choices for m, a, and c in equation (7.1.1). If we don't need a period longer than 10^4 to 10^6, say, we can keep the value of $(m-1)a+c$ small enough to avoid overflows that would otherwise mandate the extra complexity of Schrage's method (above). We can thus easily embed in our programs

```
unsigned long jran,ia,ic,im;
float ran;
...
jran=(jran*ia+ic) % im;
ran=(float) jran / (float) im;
```

whenever we want a quick and dirty uniform deviate, or

```
jran=(jran*ia+ic) % im;
j=jlo+((jhi-jlo+1)*jran)/im;
```

whenever we want an integer between `jlo` and `jhi`, inclusive. (In both cases `jran` was once initialized to any seed value between 0 and `im-1`.)

Be sure to remember, however, that when `im` is small, the kth root of it, which is the number of planes in k-space, is even smaller! So a quick and dirty generator should never be used to select points in k-space with $k > 1$.

With these caveats, some "good" choices for the constants are given in the accompanying table. These constants (i) give a period of maximal length `im`, and, more important, (ii) pass Knuth's "spectral test" for dimensions 2, 3, 4, 5, and 6. The increment `ic` is a prime, close to the value $(\frac{1}{2}-\frac{1}{6}\sqrt{3})$`im`; actually almost any value of `ic` that is relatively prime to `im` will do just as well, but there is some "lore" favoring this choice (see [4], p. 84).

An Even Quicker Generator

In `C`, if you multiply two `unsigned long int` integers on a machine with a 32-bit long integer representation, the value returned is the low-order 32 bits of the true 64-bit product. If we now choose $m = 2^{32}$, the "mod" in equation (7.1.1) is free, and we have simply

$$I_{j+1} = aI_j + c \tag{7.1.6}$$

Knuth suggests $a = 1664525$ as a suitable multiplier for this value of m. H.W. Lewis has conducted extensive tests of this value of a with $c = 1013904223$, which is a prime close to $(\sqrt{5}-2)m$. The resulting in-line generator (we will call it `ranqd1`) is simply

```
unsigned long idum;
...
idum = 1664525L*idum + 1013904223L;
```

This is about as good as any 32-bit linear congruential generator, entirely adequate for many uses. And, with only a single multiply and add, it is *very* fast.

To check whether your machine has the desired integer properties, see if you can generate the following sequence of 32-bit values (given here in hex): 00000000, 3C6EF35F, 47502932, D1CCF6E9, AAF95334, 6252E503, 9F2EC686, 57FE6C2D, A3D95FA8, 81FDBEE7, 94F0AF1A, CBF633B1.

If you need floating-point values instead of 32-bit integers, and want to avoid a divide by floating-point 2^{32}, a dirty trick is to mask in an exponent that makes the value lie between 1 and 2, then subtract 1.0. The resulting in-line generator (call it `ranqd2`) will look something like

Constants for Quick and Dirty Random Number Generators

overflow at	im	ia	ic
	6075	106	1283
2^{20}			
	7875	211	1663
2^{21}			
	7875	421	1663
2^{22}			
	6075	1366	1283
	6655	936	1399
	11979	430	2531
2^{23}			
	14406	967	3041
	29282	419	6173
	53125	171	11213
2^{24}			
	12960	1741	2731
	14000	1541	2957
	21870	1291	4621
	31104	625	6571
	139968	205	29573
2^{25}			
	29282	1255	6173
	81000	421	17117
	134456	281	28411
2^{26}			

overflow at	im	ia	ic
	86436	1093	18257
	121500	1021	25673
	259200	421	54773
2^{27}			
	117128	1277	24749
	121500	2041	25673
	312500	741	66037
2^{28}			
	145800	3661	30809
	175000	2661	36979
	233280	1861	49297
	244944	1597	51749
2^{29}			
	139968	3877	29573
	214326	3613	45289
	714025	1366	150889
2^{30}			
	134456	8121	28411
	259200	7141	54773
2^{31}			
	233280	9301	49297
	714025	4096	150889
2^{32}			

```
    unsigned long idum,itemp;
    float rand;
#ifdef vax
    static unsigned long jflone = 0x00004080;
    static unsigned long jflmsk = 0xffff007f;
#else
    static unsigned long jflone = 0x3f800000;
    static unsigned long jflmsk = 0x007fffff;
#endif
    ...
    idum = 1664525L*idum + 1013904223L;
    itemp = jflone | (jflmsk & idum);
    rand = (*(float *)&itemp)-1.0;
```

The hex constants 3F800000 and 007FFFFF are the appropriate ones for computers using the IEEE representation for 32-bit floating-point numbers (e.g., IBM PCs and most UNIX workstations). For DEC VAXes, the correct hex constants are, respectively, 00004080 and FFFF007F. Notice that the IEEE mask results in the floating-point number being constructed out of the 23 low-order bits of the integer, which is not ideal. (Your authors have tried very hard to make *almost all* of the material in this book machine and compiler independent — indeed, even programming language independent. This subsection is a rare aberration. Forgive us. Once in a great while the temptation to be *really dirty* is just irresistible.)

Relative Timings and Recommendations

Timings are inevitably machine dependent. Nevertheless the following table

is indicative of the *relative* timings, for typical machines, of the various uniform generators discussed in this section, plus `ran4` from §7.5. Smaller values in the table indicate faster generators. The generators `ranqd1` and `ranqd2` refer to the "quick and dirty" generators immediately above.

Generator	Relative Execution Time
`ran0`	$\equiv 1.0$
`ran1`	≈ 1.3
`ran2`	≈ 2.0
`ran3`	≈ 0.6
`ranqd1`	≈ 0.10
`ranqd2`	≈ 0.25
`ran4`	≈ 4.0

On balance, we recommend `ran1` for general use. It is portable, based on Park and Miller's Minimal Standard generator with an additional shuffle, and has no known (to us) flaws other than period exhaustion.

If you are generating more than 100,000,000 random numbers in a single calculation (that is, more than about 5% of `ran1`'s period), we recommend the use of `ran2`, with its much longer period.

Knuth's subtractive routine `ran3` seems to be the timing winner among portable routines. Unfortunately the subtractive method is not so well studied, and not a standard. We like to keep `ran3` in reserve for a "second opinion," substituting it when we suspect another generator of introducing unwanted correlations into a calculation.

The routine `ran4` generates *extremely* good random deviates, and has some other nice properties, but it is slow. See §7.5 for discussion.

Finally, the quick and dirty in-line generators `ranqd1` and `ranqd2` are very fast, but they are somewhat machine dependent, and at best only as good as a 32-bit linear congruential generator ever is — in our view not good enough in many situations. We would use these only in very special cases, where speed is critical.

CITED REFERENCES AND FURTHER READING:

Park, S.K., and Miller, K.W. 1988, *Communications of the ACM*, vol. 31, pp. 1192–1201. [1]

Schrage, L. 1979, *ACM Transactions on Mathematical Software*, vol. 5, pp. 132–138. [2]

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation* (New York: Springer-Verlag). [3]

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §§3.2–3.3. [4]

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 10. [5]

L'Ecuyer, P. 1988, *Communications of the ACM*, vol. 31, pp. 742–774. [6]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 10.

7.2 Transformation Method: Exponential and Normal Deviates

In the previous section, we learned how to generate random deviates with a uniform probability distribution, so that the probability of generating a number between x and $x + dx$, denoted $p(x)dx$, is given by

$$p(x)dx = \begin{cases} dx & 0 < x < 1 \\ 0 & \text{otherwise} \end{cases} \tag{7.2.1}$$

The probability distribution $p(x)$ is of course normalized, so that

$$\int_{-\infty}^{\infty} p(x)dx = 1 \tag{7.2.2}$$

Now 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

$$|p(y)dy| = |p(x)dx| \tag{7.2.3}$$

or

$$p(y) = p(x)\left|\frac{dx}{dy}\right| \tag{7.2.4}$$

Exponential Deviates

As an example, suppose that $y(x) \equiv -\ln(x)$, and that $p(x)$ is as given by equation (7.2.1) for a uniform deviate. Then

$$p(y)dy = \left|\frac{dx}{dy}\right| dy = e^{-y}dy \tag{7.2.5}$$

which is distributed exponentially. This exponential distribution occurs frequently in real problems, 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 7.2.4) that the quantity y/λ has the probability distribution $\lambda e^{-\lambda y}$.

So we have

```
#include <math.h>

float expdev(long *idum)
Returns an exponentially distributed, positive, random deviate of unit mean, using
ran1(idum) as the source of uniform deviates.
{
    float ran1(long *idum);
    float dum;

    do
        dum=ran1(idum);
    while (dum == 0.0);
    return -log(dum);
}
```

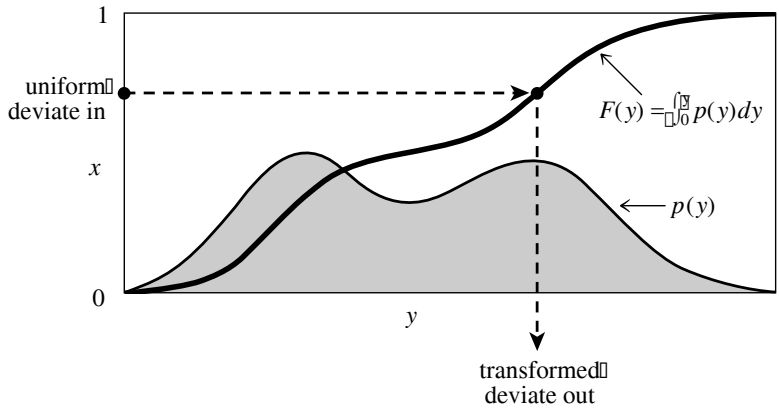

Figure 7.2.1. Transformation method for generating a random deviate y from a known probability distribution $p(y)$. The indefinite integral of $p(y)$ must be known and invertible. A uniform deviate x is chosen between 0 and 1. Its corresponding y on the definite-integral curve is the desired deviate.

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. (See Figure 7.2.1.) According to (7.2.4), we need to solve the differential equation

$$\frac{dx}{dy} = f(y) \tag{7.2.6}$$

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

$$y(x) = F^{-1}(x) \tag{7.2.7}$$

where F^{-1} is the inverse function to F. Whether (7.2.7) 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, (7.2.7) has an immediate geometric interpretation: Since $F(y)$ is the area under the probability curve to the left of y, (7.2.7) 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.

Normal (Gaussian) Deviates

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

$$p(y_1, y_2, \ldots) dy_1 dy_2 \ldots = p(x_1, x_2, \ldots) \left| \frac{\partial(x_1, x_2, \ldots)}{\partial(y_1, y_2, \ldots)} \right| dy_1 dy_2 \ldots \tag{7.2.8}$$

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

An important example of the use of (7.2.8) is the *Box-Muller* method for generating random deviates with a normal (Gaussian) distribution,

$$p(y)dy = \frac{1}{\sqrt{2\pi}} e^{-y^2/2} dy \tag{7.2.9}$$

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

$$\begin{aligned} y_1 &= \sqrt{-2\ln x_1}\cos 2\pi x_2 \\ y_2 &= \sqrt{-2\ln x_1}\sin 2\pi x_2 \end{aligned} \tag{7.2.10}$$

Equivalently we can write

$$\begin{aligned} x_1 &= \exp\left[-\frac{1}{2}(y_1^2 + y_2^2)\right] \\ x_2 &= \frac{1}{2\pi}\arctan\frac{y_2}{y_1} \end{aligned} \tag{7.2.11}$$

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

$$\frac{\partial(x_1, x_2)}{\partial(y_1, y_2)} = \begin{vmatrix} \frac{\partial x_1}{\partial y_1} & \frac{\partial x_1}{\partial y_2} \\ \frac{\partial x_2}{\partial y_1} & \frac{\partial x_2}{\partial y_2} \end{vmatrix} = -\left[\frac{1}{\sqrt{2\pi}} e^{-y_1^2/2}\right]\left[\frac{1}{\sqrt{2\pi}} e^{-y_2^2/2}\right] \tag{7.2.12}$$

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 (7.2.9).

One further trick is useful in applying (7.2.10). 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 (7.2.10) can now be written as $v_1/\sqrt{R^2}$ and $v_2/\sqrt{R^2}$, obviating the trigonometric function calls!

We thus have

```
#include <math.h>

float gasdev(long *idum)
Returns a normally distributed deviate with zero mean and unit variance, using ran1(idum)
as the source of uniform deviates.
{
    float ran1(long *idum);
    static int iset=0;
    static float gset;
    float fac,rsq,v1,v2;

    if (*idum < 0) iset=0;                  Reinitialize.
    if (iset == 0) {                        We don't have an extra deviate handy, so
        do {
            v1=2.0*ran1(idum)-1.0;          pick two uniform numbers in the square ex-
            v2=2.0*ran1(idum)-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);        and if they are not, try again.
        fac=sqrt(-2.0*log(rsq)/rsq);
        Now make the Box-Muller transformation to get two normal deviates. Return one and
        save the other for next time.
        gset=v1*fac;
        iset=1;                                    Set flag.
        return v2*fac;
    } else {                                       We have an extra deviate handy,
        iset=0;                                    so unset the flag,
        return gset;                               and return it.
    }
}
```

See Devroye [1] and Bratley [2] for many additional algorithms.

CITED REFERENCES AND FURTHER READING:

Devroye, L. 1986, *Non-Uniform Random Variate Generation* (New York: Springer-Verlag), §9.1. [1]

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

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), pp. 116ff.

7.3 Rejection Method: Gamma, Poisson, Binomial Deviates

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:

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 $f(x)$ which 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

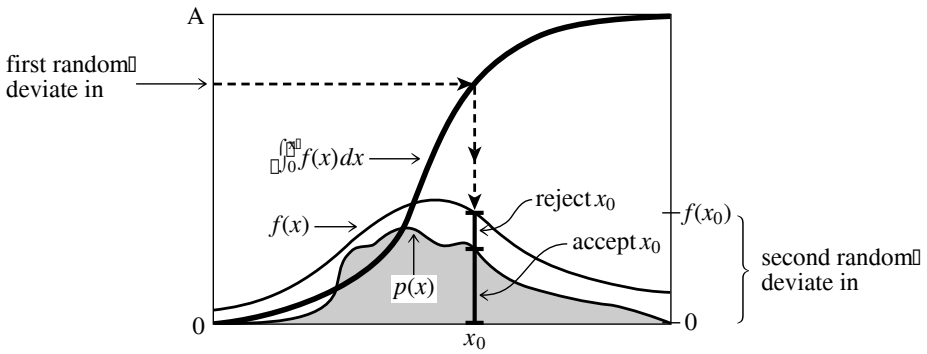

Figure 7.3.1. Rejection method for generating a random deviate x from a known probability distribution $p(x)$ that is everywhere less than some other function $f(x)$. The transformation method is first used to generate a random deviate x of the distribution f (compare Figure 7.2.1). A second uniform 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.

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 (§7.2) 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. You should be able to convince yourself that the point (x, y) is uniformly distributed in the area under the comparison function $f(x)$.

An equivalent procedure is to pick the second uniform deviate between zero and one, and accept or reject according to whether it is respectively less than or greater than the ratio $p(x)/f(x)$.

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 procedure. Then, of course, this procedure must be repeated, on the average, A times before the final deviate is obtained.

Gamma Distribution

The gamma distribution of integer order $a > 0$ is the waiting time to the ath event in a Poisson random process of unit mean. For example, when $a = 1$, it is just the exponential distribution of §7.2, the waiting time to the first event.

A gamma deviate has probability $p_a(x)dx$ of occurring with a value between x and $x + dx$, where

$$p_a(x)dx = \frac{x^{a-1}e^{-x}}{\Gamma(a)}dx \qquad x > 0 \tag{7.3.1}$$

To generate deviates of (7.3.1) for small values of a, it is best to add up a exponentially distributed waiting times, i.e., logarithms of uniform deviates. Since the sum of logarithms is the logarithm of the product, one really has only to generate the product of a uniform deviates, then take the log.

For larger values of a, the distribution (7.3.1) has a typically "bell-shaped" form, with a peak at $x = a$ and a half-width of about $\sqrt{a}$.

We will be interested in several probability distributions with this same qualitative form. A useful comparison function in such cases is derived from the *Lorentzian distribution*

$$p(y)dy = \frac{1}{\pi}\left(\frac{1}{1+y^2}\right)dy \tag{7.3.2}$$

whose inverse indefinite integral is just the tangent function. It follows that the x-coordinate of an area-uniform random point under the comparison function

$$f(x) = \frac{c_0}{1 + (x - x_0)^2/a_0^2} \tag{7.3.3}$$

for any constants a_0, c_0, and x_0, can be generated by the prescription

$$x = a_0 \tan(\pi U) + x_0 \tag{7.3.4}$$

where U is a uniform deviate between 0 and 1. Thus, for some specific "bell-shaped" $p(x)$ probability distribution, we need only find constants a_0, c_0, x_0, with the product a_0c_0 (which determines the area) as small as possible, such that (7.3.3) is everywhere greater than $p(x)$.

Ahrens has done this for the gamma distribution, yielding the following algorithm (as described in Knuth [1]):

```
#include <math.h>

float gamdev(int ia, long *idum)
Returns a deviate distributed as a gamma distribution of integer order ia, i.e., a waiting time
to the iath event in a Poisson process of unit mean, using ran1(idum) as the source of
uniform deviates.
{
    float ran1(long *idum);
    void nrerror(char error_text[]);
    int j;
    float am,e,s,v1,v2,x,y;

    if (ia < 1) nrerror("Error in routine gamdev");
    if (ia < 6) {                                   Use direct method, adding waiting
        x=1.0;                                          times.
        for (j=1;j<=ia;j++) x *= ran1(idum);
        x = -log(x);
    } else {                                        Use rejection method.
```

```
        do {
            do {
                do {
                    v1=ran1(idum);
                    v2=2.0*ran1(idum)-1.0;
                } while (v1*v1+v2*v2 > 1.0);
                y=v2/v1;
                am=ia-1;
                s=sqrt(2.0*am+1.0);
                x=s*y+am;
            } while (x <= 0.0);
            e=(1.0+y*y)*exp(am*log(x/am)-s*y);
        } while (ran1(idum) > e);
    }
    return x;
}
```

These four lines generate the tangent of a random angle, i.e., they are equivalent to y = tan(π * ran1(idum)).

We decide whether to reject x: Reject in region of zero probability. Ratio of prob. fn. to comparison fn. Reject on basis of a second uniform deviate.

Poisson Deviates

The Poisson distribution is conceptually related to the gamma distribution. It gives the probability of a certain integer number m of unit rate Poisson random events occurring in a given interval of time x, while the gamma distribution was the probability of waiting time between x and $x + dx$ to the mth event. Note that m takes on only integer values ≥ 0, so that the Poisson distribution, viewed as a continuous distribution function $p_x(m)dm$, is zero everywhere except where m is an integer ≥ 0. At such places, it is infinite, such that the integrated probability over a region containing the integer is some finite number. The total probability at an integer j is

$$\text{Prob}(j) = \int_{j-\epsilon}^{j+\epsilon} p_x(m)dm = \frac{x^j e^{-x}}{j!} \tag{7.3.5}$$

At first sight this might seem an unlikely candidate distribution for the rejection method, since no continuous comparison function can be larger than the infinitely tall, but infinitely narrow, *Dirac delta functions* in $p_x(m)$. However, there is a trick that we can do: Spread the finite area in the spike at j uniformly into the interval between j and $j + 1$. This defines a continuous distribution $q_x(m)dm$ given by

$$q_x(m)dm = \frac{x^{[m]} e^{-x}}{[m]!}dm \tag{7.3.6}$$

where $[m]$ represents the largest integer less than m. If we now use the rejection method to generate a (noninteger) deviate from (7.3.6), and then take the integer part of that deviate, it will be as if drawn from the desired distribution (7.3.5). (See Figure 7.3.2.) This trick is general for any integer-valued probability distribution.

For x large enough, the distribution (7.3.6) is qualitatively bell-shaped (albeit with a bell made out of small, square steps), and we can use the same kind of Lorentzian comparison function as was already used above. For small x, we can generate independent exponential deviates (waiting times between events); when the sum of these first exceeds x, then the number of events that would have occurred in waiting time x becomes known and is one less than the number of terms in the sum.

These ideas produce the following routine:

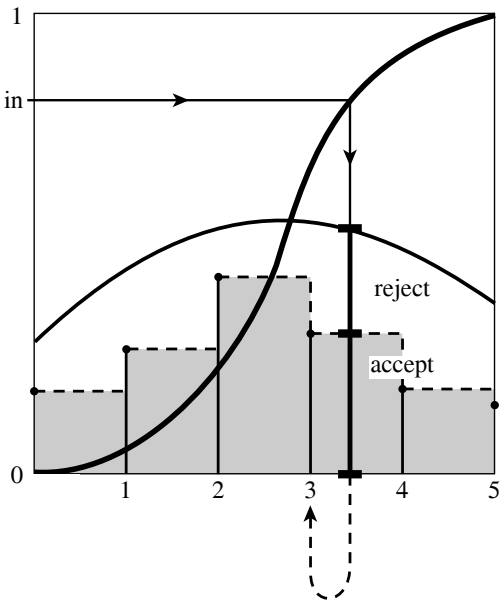

Figure 7.3.2. Rejection method as applied to an integer-valued distribution. The method is performed on the step function shown as a dashed line, yielding a real-valued deviate. This deviate is rounded down to the next lower integer, which is output.

```
#include <math.h>
#define PI 3.141592654

float poidev(float xm, long *idum)
Returns as a floating-point number an integer value that is a random deviate drawn from a
Poisson distribution of mean xm, using ran1(idum) as a source of uniform random deviates.
{
    float gammln(float xx);
    float ran1(long *idum);
    static float sq,alxm,g,oldm=(-1.0);        oldm is a flag for whether xm has changed
    float em,t,y;                                  since last call.

    if (xm < 12.0) {                       Use direct method.
        if (xm != oldm) {
            oldm=xm;
            g=exp(-xm);                    If xm is new, compute the exponential.
        }
        em = -1;
        t=1.0;
        do {                               Instead of adding exponential deviates it is equiv-
            ++em;                              alent to multiply uniform deviates. We never
            t *= ran1(idum);                   actually have to take the log, merely com-
        } while (t > g);                       pare to the pre-computed exponential.
    } else {                               Use rejection method.
        if (xm != oldm) {                  If xm has changed since the last call, then pre-
            oldm=xm;                           compute some functions that occur below.
            sq=sqrt(2.0*xm);
            alxm=log(xm);
            g=xm*alxm-gammln(xm+1.0);
            The function gammln is the natural log of the gamma function, as given in §6.1.
        }
        do {
            do {                           y is a deviate from a Lorentzian comparison func-
                y=tan(PI*ran1(idum));          tion.
```

```
            em=sq*y+xm;                 em is y, shifted and scaled.
        } while (em < 0.0);             Reject if in regime of zero probability.
        em=floor(em);                   The trick for integer-valued distributions.
        t=0.9*(1.0+y*y)*exp(em*alxm-gammln(em+1.0)-g);
        The ratio of the desired distribution to the comparison function; we accept or
        reject by comparing it to another uniform deviate. The factor 0.9 is chosen so
        that t never exceeds 1.
      } while (ran1(idum) > t);
    }
    return em;
}
```

Binomial Deviates

If an event occurs with probability q, and we make n trials, then the number of times m that it occurs has the binomial distribution,

$$\int_{j-\epsilon}^{j+\epsilon} p_{n,q}(m)dm = \binom{n}{j} q^j (1-q)^{n-j} \tag{7.3.7}$$

The binomial distribution is integer valued, with m taking on possible values from 0 to n. It depends on *two* parameters, n and q, so is correspondingly a bit harder to implement than our previous examples. Nevertheless, the techniques already illustrated are sufficiently powerful to do the job:

```
#include <math.h>
#define PI 3.141592654

float bnldev(float pp, int n, long *idum)
Returns as a floating-point number an integer value that is a random deviate drawn from
a binomial distribution of n trials each of probability pp, using ran1(idum) as a source of
uniform random deviates.
{
    float gammln(float xx);
    float ran1(long *idum);
    int j;
    static int nold=(-1);
    float am,em,g,angle,p,bnl,sq,t,y;
    static float pold=(-1.0),pc,plog,pclog,en,oldg;

    p=(pp <= 0.5 ? pp : 1.0-pp);
    The binomial distribution is invariant under changing pp to 1-pp, if we also change the
    answer to n minus itself; we'll remember to do this below.
    am=n*p;                             This is the mean of the deviate to be produced.
    if (n < 25) {                       Use the direct method while n is not too large.
        bnl=0.0;                            This can require up to 25 calls to ran1.
        for (j=1;j<=n;j++)
            if (ran1(idum) < p) ++bnl;
    } else if (am < 1.0) {              If fewer than one event is expected out of 25
        g=exp(-am);                         or more trials, then the distribution is quite
        t=1.0;                              accurately Poisson. Use direct Poisson method.
        for (j=0;j<=n;j++) {
            t *= ran1(idum);
            if (t < g) break;
        }
        bnl=(j <= n ? j : n);
    } else {                            Use the rejection method.
```

```
        if (n != nold) {                    If n has changed, then compute useful quanti-
            en=n;                               ties.
            oldg=gammln(en+1.0);
            nold=n;
        } if (p != pold) {                  If p has changed, then compute useful quanti-
            pc=1.0-p;                           ties.
            plog=log(p);
            pclog=log(pc);
            pold=p;
        }
        sq=sqrt(2.0*am*pc);                 The following code should by now seem familiar:
        do {                                    rejection method with a Lorentzian compar-
            do {                                ison function.
                angle=PI*ran1(idum);
                y=tan(angle);
                em=sq*y+am;
            } while (em < 0.0 || em >= (en+1.0));       Reject.
            em=floor(em);                   Trick for integer-valued distribution.
            t=1.2*sq*(1.0+y*y)*exp(oldg-gammln(em+1.0)
                -gammln(en-em+1.0)+em*plog+(en-em)*pclog);
        } while (ran1(idum) > t);           Reject. This happens about 1.5 times per devi-
        bnl=em;                                 ate, on average.
    }
    if (p != pp) bnl=n-bnl;                 Remember to undo the symmetry transforma-
    return bnl;                                 tion.
}
```

See Devroye [2] and Bratley [3] for many additional algorithms.

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), pp. 120ff. [1]

Devroye, L. 1986, *Non-Uniform Random Variate Generation* (New York: Springer-Verlag), §X.4. [2]

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation* (New York: Springer-Verlag). [3].

7.4 Generation of Random Bits

The C language gives you useful access to some machine-level bitwise operations such as << (left shift). This section will show you how to put such abilities to good use.

The problem is how to generate single random bits, with 0 and 1 equally probable. Of course you can just generate uniform random deviates between zero and one and use their high-order bit (i.e., test if they are greater than or less than 0.5). However this takes a lot of arithmetic; there are special-purpose applications, such as real-time signal processing, where you want to generate bits very much faster than that.

One method for generating random bits, with two variant implementations, is based on "primitive polynomials modulo 2." The theory of these polynomials is beyond our scope (although §7.7 and §20.3 will give you small tastes of it). Here,

suffice it to say that there are special polynomials among those whose coefficients are zero or one. An example is

$$x^{18} + x^5 + x^2 + x^1 + x^0 \tag{7.4.1}$$

which we can abbreviate by just writing the nonzero powers of x, e.g.,

$$(18, 5, 2, 1, 0)$$

Every primitive polynomial modulo 2 of order n (=18 above) defines a recurrence relation for obtaining a new random bit from the n preceding ones. The recurrence relation is guaranteed to produce a sequence of maximal length, i.e., cycle through all possible sequences of n bits (except all zeros) before it repeats. Therefore one can seed the sequence with any initial bit pattern (except all zeros), and get $2^n - 1$ random bits before the sequence repeats.

Let the bits be numbered from 1 (most recently generated) through n (generated n steps ago), and denoted $a_1, a_2, \ldots, a_n$. We want to give a formula for a new bit a_0. After generating a_0 we will shift all the bits by one, so that the old a_n is finally lost, and the new a_0 becomes a_1. We then apply the formula again, and so on.

"Method I" is the easiest to implement in hardware, requiring only a single shift register n bits long and a few XOR ("exclusive or" or bit addition mod 2) gates, the operation denoted in C by "$\wedge$". For the primitive polynomial given above, the recurrence formula is

$$a_0 = a_{18} \wedge a_5 \wedge a_2 \wedge a_1 \tag{7.4.2}$$

The terms that are $\wedge$'d together can be thought of as "taps" on the shift register, $\wedge$'d into the register's input. More generally, there is precisely one term for each nonzero coefficient in the primitive polynomial except the constant (zero bit) term. So the first term will always be a_n for a primitive polynomial of degree n, while the last term might or might not be a_1, depending on whether the primitive polynomial has a term in x^1.

While it is simple in hardware, Method I is somewhat cumbersome in C, because the individual bits must be collected by a sequence of full-word masks:

```
int irbit1(unsigned long *iseed)
Returns as an integer a random bit, based on the 18 low-significance bits in iseed (which is
modified for the next call).
{
    unsigned long newbit;                   The accumulated XOR's.

    newbit = (*iseed >> 17) & 1             Get bit 18.
        ^ (*iseed >> 4) & 1                 XOR with bit 5.
        ^ (*iseed >> 1) & 1                 XOR with bit 2.
        ^ (*iseed & 1);                     XOR with bit 1.
    *iseed=(*iseed << 1) | newbit;          Leftshift the seed and put the result of the
    return (int) newbit;                        XOR's in its bit 1.
}
```

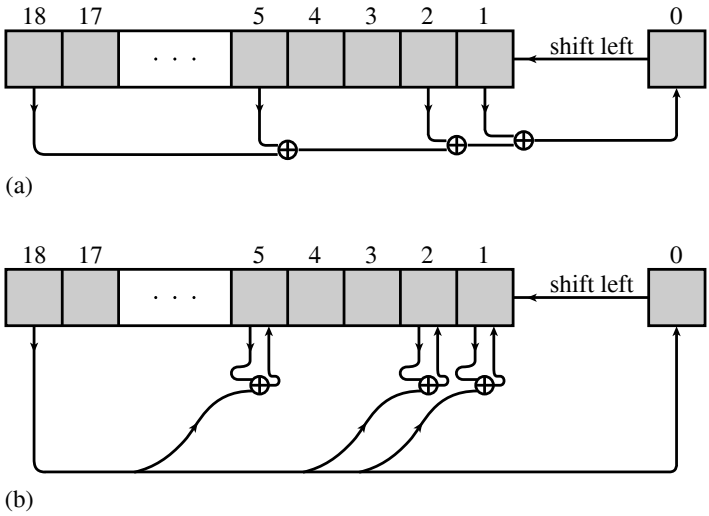

Figure 7.4.1. Two related methods for obtaining random bits from a shift register and a primitive polynomial modulo 2. (a) The contents of selected taps are combined by exclusive-or (addition modulo 2), and the result is shifted in from the right. This method is easiest to implement in hardware. (b) Selected bits are modified by exclusive-or with the leftmost bit, which is then shifted in from the right. This method is easiest to implement in software.

"Method II" is less suited to direct hardware implementation (though still possible), but is beautifully suited to C. It modifies more than one bit among the saved n bits as each new bit is generated (Figure 7.4.1). It generates the maximal length sequence, but not in the same order as Method I. The prescription for the primitive polynomial (7.4.1) is:

$$\begin{aligned} a_0 &= a_{18} \\ a_5 &= a_5 \wedge a_0 \\ a_2 &= a_2 \wedge a_0 \\ a_1 &= a_1 \wedge a_0 \end{aligned} \tag{7.4.3}$$

In general there will be an exclusive-or for each nonzero term in the primitive polynomial except 0 and n. The nice feature about Method II is that all the exclusive-or's can usually be done as a single full-word exclusive-or operation:

```
#define IB1 1                       Powers of 2.
#define IB2 2
#define IB5 16
#define IB18 131072
#define MASK (IB1+IB2+IB5)

int irbit2(unsigned long *iseed)
Returns as an integer a random bit, based on the 18 low-significance bits in iseed (which is
modified for the next call).
{
    if (*iseed & IB18) {            Change all masked bits, shift, and put 1 into bit 1.
        *iseed=((*iseed ^ MASK) << 1) | IB1;
        return 1;
    } else {                        Shift and put 0 into bit 1.
```

```
        *iseed <<= 1;
        return 0;
    }
}
```

Some Primitive Polynomials Modulo 2 (after Watson)

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

A word of caution is: Don't use sequential bits from these routines as the bits of a large, supposedly random, integer, or as the bits in the mantissa of a supposedly

random floating-point number. They are not very random for that purpose; see Knuth [1]. Examples of acceptable uses of these random bits are: (i) multiplying a signal randomly by ± 1 at a rapid "chip rate," so as to spread its spectrum uniformly (but recoverably) across some desired bandpass, or (ii) Monte Carlo exploration of a binary tree, where decisions as to whether to branch left or right are to be made randomly.

Now we do not want you to go through life thinking that there is something special about the primitive polynomial of degree 18 used in the above examples. (We chose 18 because 2^{18} is small enough for you to verify our claims directly by numerical experiment.) The accompanying table [2] lists one primitive polynomial for each degree up to 100. (In fact there exist many such for each degree. For example, see §7.7 for a complete table up to degree 10.)

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), pp. 29ff. [1]

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

Tausworthe, R.C. 1965, *Mathematics of Computation*, vol. 19, pp. 201–209.

Watson, E.J. 1962, *Mathematics of Computation*, vol. 16, pp. 368–369. [2]

7.5 Random Sequences Based on Data Encryption

In *Numerical Recipes'* first edition, we described how to use the Data Encryption Standard (DES) [1-3] for the generation of random numbers. Unfortunately, when implemented in software in a high-level language like C, DES is very slow, so excruciatingly slow, in fact, that our previous implementation can be viewed as more mischievous than useful. Here we give a much faster and simpler algorithm which, though it may not be secure in the cryptographic sense, generates about equally good random numbers.

DES, like its progenitor cryptographic system LUCIFER, is a so-called "block product cipher" [4]. It acts on 64 bits of input by iteratively applying (16 times, in fact) a kind of highly nonlinear bit-mixing function. Figure 7.5.1 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 [4] 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. Apparently, this function was chosen to be particularly strong cryptographically (or conceivably as some critics contend, to have an exquisitely subtle cryptographic flaw!). For our purposes, a different function g that can be rapidly computed in a high-level computer language is preferable. Such a function may weaken the algorithm cryptographically. Our purposes are not, however, cryptographic: We want to find the fastest g, and smallest number of iterations of the mixing procedure in Figure 7.5.1, such that our output random sequence passes the standard tests that are customarily applied to random number generators. The resulting algorithm will not be DES, but rather a kind of "pseudo-DES," better suited to the purpose at hand.

Following the criterion, mentioned above, that g should be nonlinear, we must give the integer multiply operation a prominent place in g. Because 64-bit registers are not generally accessible in high-level languages, we must confine ourselves to multiplying 16-bit operands

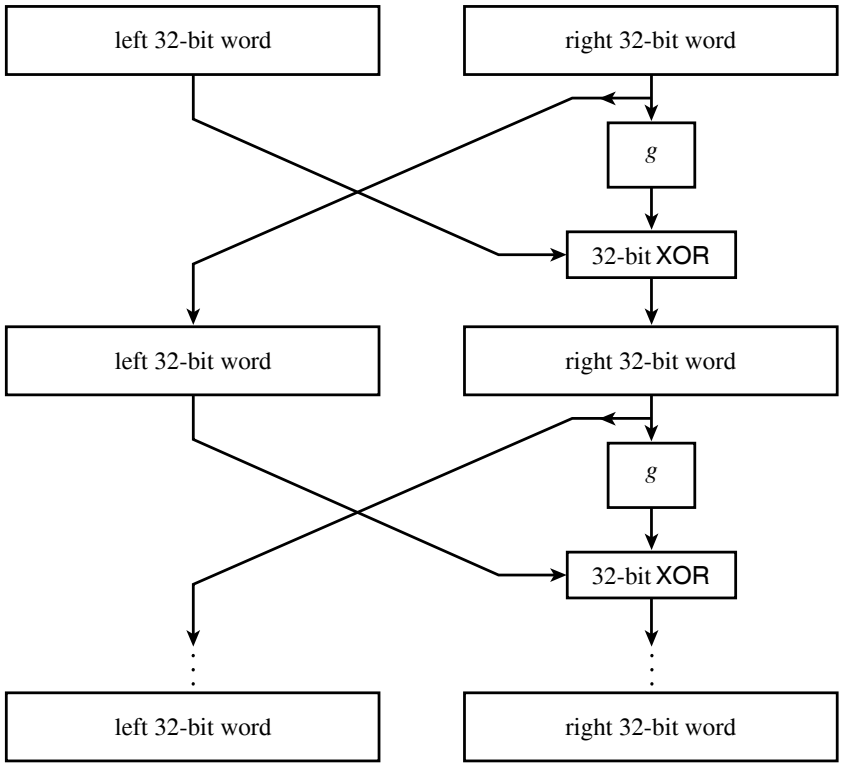

Figure 7.5.1. The Data Encryption Standard (DES) iterates a nonlinear function g on two 32-bit words, in the manner shown here (after Meyer and Matyas [4]).

into a 32-bit result. So, the general idea of g, almost forced, 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 7.5.2. 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 towards 0 or 1 bits.

The "reverse half-words" operation in Figure 7.5.2 turns out to be essential; otherwise, the very lowest and very highest bits are not properly mixed by the three multiplications. The nonobvious choices in g are therefore: where along the vertical "pipeline" to do the reverse; in what order to combine the three products and C_2; and with which operation (add or exclusive-or) should each combining be done? We tested these choices exhaustively before settling on the algorithm shown in the figure.

It remains to determine the smallest number of iterations N_{it} that we can get away with. The minimum meaningful N_{it} is evidently two, since a single iteration simply moves one 32-bit word without altering it. One can use the constants C_1 and C_2 to help determine an appropriate N_{it}: When $N_{it} = 2$ and $C_1 = C_2 = 0$ (an intentionally very poor choice), the generator fails several tests of randomness by easily measurable, though not overwhelming, amounts. When $N_{it} = 4$, on the other hand, or with $N_{it} = 2$ but with the constants C_1, C_2 nonsparse, we have been unable to find *any* statistical deviation from randomness in sequences of up to 10^9 floating numbers r_i derived from this scheme. The combined strength of $N_{it} = 4$ and nonsparse C_1, C_2 should therefore give sequences that are random to tests even far beyond those that we have actually tried. These are our recommended conservative

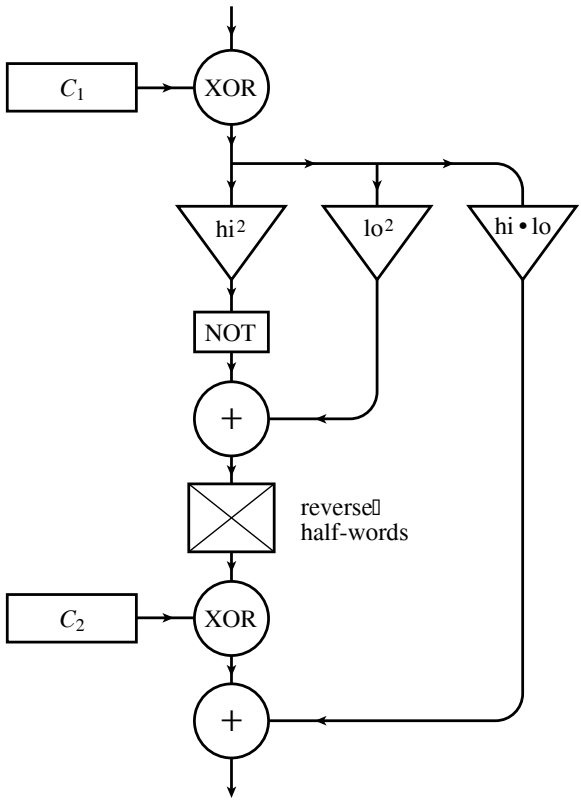

Figure 7.5.2. The nonlinear function g used by the routine `psdes`.

parameter values, notwithstanding the fact that $N_{it} = 2$ (which is, of course, twice as fast) has no nonrandomness discernible (by us).

Implementation of these ideas is straightforward. The following routine is not quite strictly portable, since it assumes that `unsigned long` integers are 32-bits, as is the case on most machines. However, there is no reason to believe that *longer* integers would be in any way inferior (with suitable extensions of the constants C_1, C_2). `C` does not provide a convenient, portable way to divide a long integer into half words, so we must use a combination of masking (`& 0xffff`) with left- and right-shifts by 16 bits (`<<16` and `>>16`). On some machines the half-word extraction could be made faster by the use of `C`'s union construction, but this would generally not be portable between "big-endian" and "little-endian" machines. (Big- and little-endian refer to the order in which the bytes are stored in a word.)

```
#define NITER 4

void psdes(unsigned long *lword, unsigned long *irword)
"Pseudo-DES" hashing of the 64-bit word (lword,irword). Both 32-bit arguments are re-
turned hashed on all bits.
{
    unsigned long i,ia,ib,iswap,itmph=0,itmpl=0;
    static unsigned long c1[NITER]={
        0xbaa96887L, 0x1e17d32cL, 0x03bcdc3cL, 0x0f33d1b2L};
    static unsigned long c2[NITER]={
        0x4b0f3b58L, 0xe874f0c3L, 0x6955c5a6L, 0x55a7ca46L};

    for (i=0;i<NITER;i++) {
    Perform niter iterations of DES logic, using a simpler (non-cryptographic) nonlinear func-
    tion instead of DES's.
```

```
        ia=(iswap=(*irword)) ^ c1[i];          The bit-rich constants c1 and (below)
        itmpl = ia & 0xffff;                       c2 guarantee lots of nonlinear mix-
        itmph = ia >> 16;                          ing.
        ib=itmpl*itmpl+ ~(itmph*itmph);
        *irword=(*lword) ^ (((ia = (ib >> 16) |
            ((ib & 0xffff) << 16)) ^ c2[i])+itmpl*itmph);
        *lword=iswap;
    }
}
```

The routine `ran4`, listed below, uses `psdes` to generate uniform random deviates. We adopt the convention that a negative value of the argument `idum` sets the left 32-bit word, while a positive value i sets the right 32-bit word, returns the ith random deviate, and increments `idum` to $i + 1$. This is no more than a convenient way of defining many different sequences (negative values of `idum`), but still with random access to each sequence (positive values of `idum`). For getting a floating-point number from the 32-bit integer, we like to do it by the masking trick described at the end of §7.1, above. The hex constants 3F800000 and 007FFFFF are the appropriate ones for computers using the IEEE representation for 32-bit floating-point numbers (e.g., IBM PCs and most UNIX workstations). For DEC VAXes, the correct hex constants are, respectively, 00004080 and FFFF007F. For greater portability, you can instead construct a floating number by making the (signed) 32-bit integer nonnegative (typically, you add exactly 2^{31} if it is negative) and then multiplying it by a floating constant (typically $2.^{-31}$).

An interesting, and sometimes useful, feature of the routine `ran4`, below, is that it allows random access to the nth random value in a sequence, without the necessity of first generating values $1 \cdots n-1$. This property is shared by any random number generator based on *hashing* (the technique of mapping data keys, which may be highly clustered in value, approximately uniformly into a storage address space) [5,6]. One might have a simulation problem in which some certain rare situation becomes recognizable by its consequences only considerably after it has occurred. One may wish to restart the simulation back at that occurrence, using identical random values but, say, varying some other control parameters. The relevant question might then be something like "what random numbers were used in cycle number 337098901?" It might already be cycle number 395100273 before the question comes up. Random generators based on recursion, rather than hashing, cannot easily answer such a question.

`float ran4(long *idum)`
Returns a uniform random deviate in the range 0.0 to 1.0, generated by pseudo-DES (DES-like) hashing of the 64-bit word `(idums,idum)`, where `idums` was set by a previous call with negative `idum`. Also increments `idum`. Routine can be used to generate a random sequence by successive calls, leaving `idum` unaltered between calls; or it can randomly access the nth deviate in a sequence by calling with `idum` $= n$. Different sequences are initialized by calls with differing negative values of `idum`.

```
{
    void psdes(unsigned long *lword, unsigned long *irword);
    unsigned long irword,itemp,lword;
    static long idums = 0;
    The hexadecimal constants jflone and jflmsk below are used to produce a floating number
    between 1. and 2. by bitwise masking. They are machine-dependent. See text.
#if defined(vax) || defined(_vax_) || defined(__vax__) || defined(VAX)
    static unsigned long jflone = 0x00004080;
    static unsigned long jflmsk = 0xffff007f;
#else
    static unsigned long jflone = 0x3f800000;
    static unsigned long jflmsk = 0x007fffff;
#endif

    if (*idum < 0) {                          Reset idums and prepare to return the first
        idums = -(*idum);                         deviate in its sequence.
        *idum=1;
    }
    irword=(*idum);
    lword=idums;
```

```
    psdes(&lword,&irword);                  "Pseudo-DES" encode the words.
    itemp=jflone | (jflmsk & irword);       Mask to a floating number between 1 and
    ++(*idum);                                  2.
    return (*(float *)&itemp)-1.0;          Subtraction moves range to 0. to 1.
}
```

The accompanying table gives data for verifying that `ran4` and `psdes` work correctly on your machine. We do not advise the use of `ran4` unless you are able to reproduce the hex values shown. Typically, `ran4` is about 4 times slower than `ran0` (§7.1), or about 3 times slower than `ran1`.

Values for Verifying the Implementation of `psdes`

`idum`	before `psdes` call		after `psdes` call (hex)		`ran4(idum)`	
	`lword`	`irword`	`lword`	`irword`	VAX	PC
–1	1	1	604D1DCE	509C0C23	0.275898	0.219120
99	1	99	D97F8571	A66CB41A	0.208204	0.849246
–99	99	1	7822309D	64300984	0.034307	0.375290
99	99	99	D7F376F0	59BA89EB	0.838676	0.457334

Successive calls to `ran4` with arguments -1, 99, -99, and 99 should produce exactly the `lword` and `irword` values shown. Masking conversion to a returned floating random value is allowed to be machine dependent; values for VAX and PC are shown.

CITED REFERENCES AND FURTHER READING:

Data Encryption Standard, 1977 January 15, Federal Information Processing Standards Publication, number 46 (Washington: U.S. Department of Commerce, National Bureau of Standards). [1]

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). [2]

Validating the Correctness of Hardware Implementations of the NBS Data Encryption Standard, 1980, NBS Special Publication 500–20 (Washington: U.S. Department of Commerce, National Bureau of Standards). [3]

Meyer, C.H. and Matyas, S.M. 1982, *Cryptography: A New Dimension in Computer Data Security* (New York: Wiley). [4]

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), Chapter 6. [5]

Vitter, J.S., and Chen, W-C. 1987, *Design and Analysis of Coalesced Hashing* (New York: Oxford University Press). [6]

7.6 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 see in §10.9) 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 $x_1, \ldots, x_N$. Then the basic theorem of Monte Carlo integration estimates the integral of a function f over the multidimensional volume,

$$\int f \, dV \approx V \langle f \rangle \pm V \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}} \tag{7.6.1}$$

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

$$\langle f \rangle \equiv \frac{1}{N} \sum_{i=1}^{N} f(x_i) \qquad \langle f^2 \rangle \equiv \frac{1}{N} \sum_{i=1}^{N} f^2(x_i) \tag{7.6.2}$$

The "plus-or-minus" term in (7.6.1) 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 (Figure 7.6.1), 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 (7.6.1). 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 (7.6.1) takes this into account.

General purpose routines for Monte Carlo integration are quite complicated (see §7.8), but 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 complicated shape, namely the intersection of a torus with the edge of a large box. In particular let the object be defined by the three simultaneous conditions

$$z^2 + \left(\sqrt{x^2 + y^2} - 3 \right)^2 \le 1 \tag{7.6.3}$$

(torus centered on the origin with major radius $= 4$, minor radius $= 2$)

$$x \ge 1 \qquad y \ge -3 \tag{7.6.4}$$

(two faces of the box, see Figure 7.6.2). Suppose for the moment that the object has a constant density ρ.

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

$$\int \rho \, dx \, dy \, dz \qquad \int x\rho \, dx \, dy \, dz \qquad \int y\rho \, dx \, dy \, dz \qquad \int z\rho \, dx \, dy \, dz \tag{7.6.5}$$

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

In the following fragment, the region V, enclosing the piece-of-torus W, is the rectangular box extending from 1 to 4 in x, -3 to 4 in y, and -1 to 1 in z.

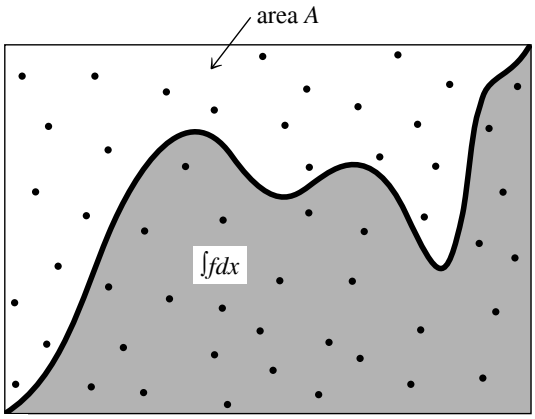

Figure 7.6.1. Monte Carlo integration. Random points are chosen within the area A. The integral of the function f is estimated as the area of A multiplied by the fraction of random points that fall below the curve f. Refinements on this procedure can improve the accuracy of the method; see text.

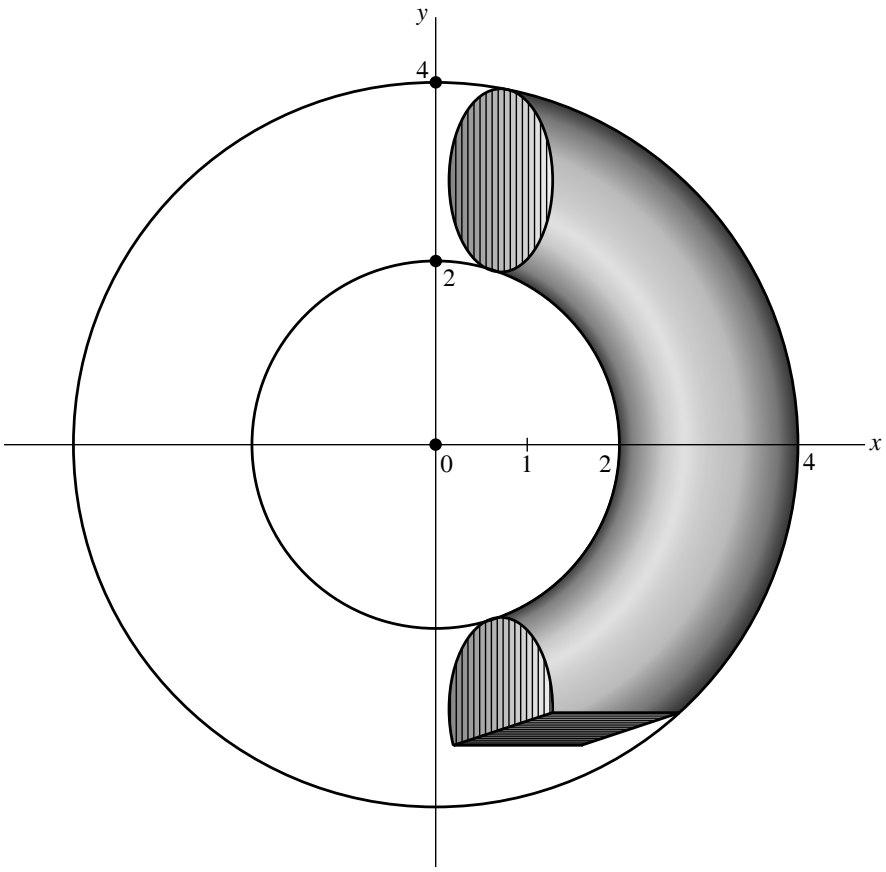

Figure 7.6.2. Example of Monte Carlo integration (see text). The region of interest is a piece of a torus, bounded by the intersection of two planes. The limits of integration of the region cannot easily be written in analytically closed form, so Monte Carlo is a useful technique.

```
#include "nrutil.h"
...
n=...                                   Set to the number of sample points desired.
den=...                                 Set to the constant value of the density.
sw=swx=swy=swz=0.0;                     Zero the various sums to be accumulated.
varw=varx=vary=varz=0.0;
vol=3.0*7.0*2.0;                        Volume of the sampled region.
for(j=1;j<=n;j++) {
    x=1.0+3.0*ran2(&idum);              Pick a point randomly in the sampled re-
    y=(-3.0)+7.0*ran2(&idum);               gion.
    z=(-1.0)+2.0*ran2(&idum);
    if (z*z+SQR(sqrt(x*x+y*y)-3.0) < 1.0) {     Is it in the torus?
        sw += den;                      If so, add to the various cumulants.
        swx += x*den;
        swy += y*den;
        swz += z*den;
        varw += SQR(den);
        varx += SQR(x*den);
        vary += SQR(y*den);
        varz += SQR(z*den);
    }
}
w=vol*sw/n;                             The values of the integrals (7.6.5),
x=vol*swx/n;
y=vol*swy/n;
z=vol*swz/n;
dw=vol*sqrt((varw/n-SQR(sw/n))/n);      and their corresponding error estimates.
dx=vol*sqrt((varx/n-SQR(swx/n))/n);
dy=vol*sqrt((vary/n-SQR(swy/n))/n);
dz=vol*sqrt((varz/n-SQR(swz/n))/n);
```

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

$$\rho(x, y, z) = e^{5z} \tag{7.6.6}$$

One way to do this is to put the statement

```
den=exp(5.0*z);
```

inside the `if (...)` block, just before `den` is first used. This will work, but it is a poor way to proceed. Since (7.6.6) 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 §7.2, solves this problem. Let

$$ds = e^{5z} dz \qquad \text{so that} \qquad s = \frac{1}{5} e^{5z}, \quad z = \frac{1}{5} \ln(5s) \tag{7.6.7}$$

Then $\rho dz = ds$, and the limits $-1 < z < 1$ become $.00135 < s < 29.682$. The program fragment now looks like this

```
#include "nrutil.h"
...
n=...                                   Set to the number of sample points desired.
sw=swx=swy=swz=0.0;
varw=varx=vary=varz=0.0;
ss=0.2*(exp(5.0)-exp(-5.0))             Interval of s to be random sampled.
vol=3.0*7.0*ss                          Volume in x,y,s-space.
for(j=1;j<=n;j++) {
    x=1.0+3.0*ran2(&idum);
    y=(-3.0)+7.0*ran2(&idum);
    s=0.00135+ss*ran2(&idum);           Pick a point in s.
    z=0.2*log(5.0*s);                   Equation (7.6.7).
    if (z*z+SQR(sqrt(x*x+y*y)-3.0) < 1.0) {
        sw += 1.0;                      Density is 1, since absorbed into definition
        swx += x;                           of s.
        swy += y;
        swz += z;
        varw += 1.0;
        varx += x*x;
        vary += y*y;
        varz += z*z;
    }
}
w=vol*sw/n;                             The values of the integrals (7.6.5),
x=vol*swx/n;
y=vol*swy/n;
z=vol*swz/n;
dw=vol*sqrt((varw/n-SQR(sw/n))/n);      and their corresponding error estimates.
dx=vol*sqrt((varx/n-SQR(swx/n))/n);
dy=vol*sqrt((vary/n-SQR(swy/n))/n);
dz=vol*sqrt((varz/n-SQR(swz/n))/n);
```

If you think for a minute, you will realize that equation (7.6.7) 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 §7.2.) 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 (7.6.1) 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 budget is large, then the technique is highly recommended as one of great generality. In the next two sections 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.

CITED REFERENCES AND FURTHER READING:
Hammersley, J.M., and Handscomb, D.C. 1964, *Monte Carlo Methods* (London: Methuen).
Shreider, Yu. A. (ed.) 1966, *The Monte Carlo Method* (Oxford: Pergamon).
Sobol', I.M. 1974, *The Monte Carlo Method* (Chicago: University of Chicago Press).
Kalos, M.H., and Whitlock, P.A. 1986, *Monte Carlo Methods* (New York: Wiley).

7.7 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 7.6.2). 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

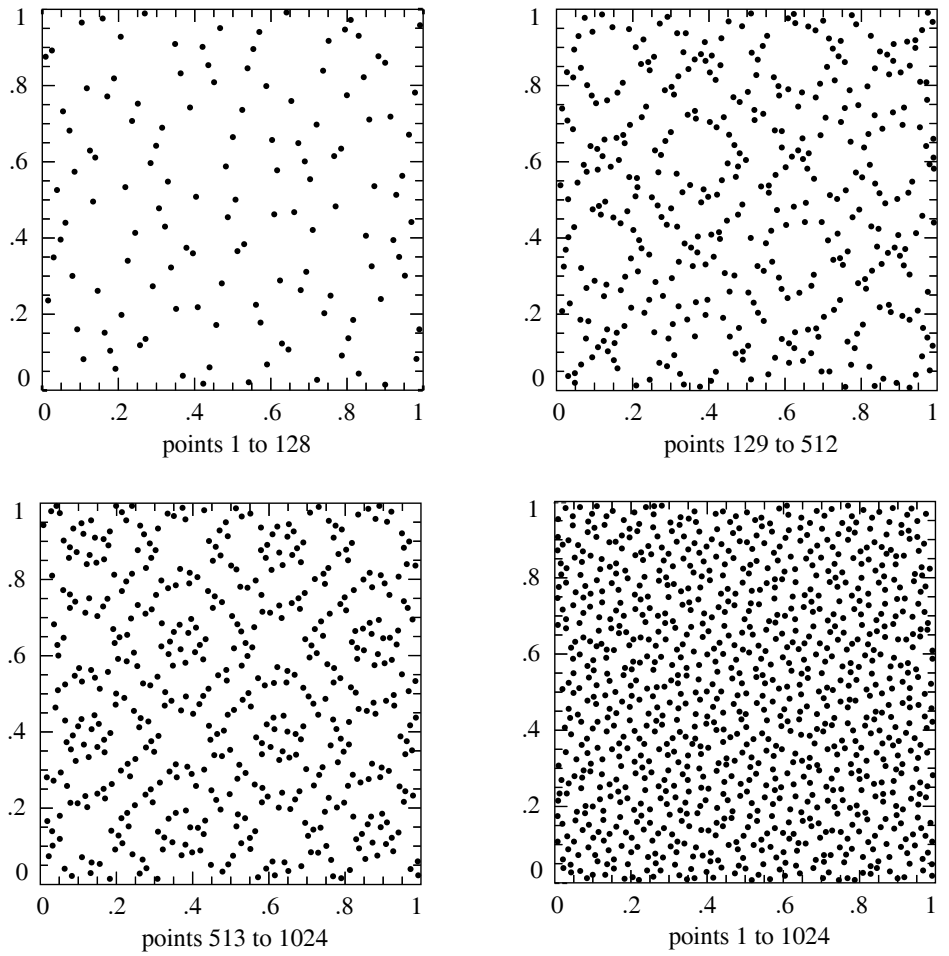

Figure 7.7.1. First 1024 points of a two-dimensional Sobol' sequence. The sequence is generated number-theoretically, rather than randomly, so successive points at any stage "know" how to fill in the gaps in the previously generated distribution.

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 suggested 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.

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 representing the interior bits (that is, omitting the high-order bit and the unit bit).	

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 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 §20.2.)

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 7.7.1 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 ourself with a cookbook summary only: Each different Sobol' sequence (or component of an n-dimensional 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 §7.4, and are further discussed in §20.3.) Suppose P is such a polynomial, of degree q,

$$P = x^q + a_1 x^{q-1} + a_2 x^{q-2} + \cdots + a_{q-1} x + 1 \tag{7.7.1}$$

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.					

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

$$M_i = 2a_1 M_{i-1} \oplus 2^2 a_2 M_{i-2} \oplus \cdots \oplus 2^{q-1} M_{i-q+1} a_{q-1} \oplus (2^q M_{i-q} \oplus M_{i-q}) \quad (7.7.2)$$

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

$$V_i = M_i/2^i \qquad i = 1, \ldots, w \quad (7.7.3)$$

The accompanying table 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), `mdeg` (their degrees), and `iv` (the starting values for the recurrence, equation 7.7.2). A second table, above, elucidates the initializing data in the routine.

```
#include "nrutil.h"
#define MAXBIT 30
#define MAXDIM 6

void sobseq(int *n, float x[])
When n is negative, internally initializes a set of MAXBIT direction numbers for each of MAXDIM
different Sobol' sequences. When n is positive (but ≤MAXDIM), returns as the vector x[1..n]
the next values from n of these sequences. (n must not be changed between initializations.)
{
    int j,k,l;
    unsigned long i,im,ipp;
    static float fac;
    static unsigned long in,ix[MAXDIM+1],*iu[MAXBIT+1];
    static unsigned long mdeg[MAXDIM+1]={0,1,2,3,3,4,4};
    static unsigned long ip[MAXDIM+1]={0,0,1,1,2,1,4};
    static unsigned long iv[MAXDIM*MAXBIT+1]={
        0,1,1,1,1,1,1,3,1,3,3,1,1,5,7,7,3,3,5,15,11,5,15,13,9};

    if (*n < 0) {                                   Initialize, don't return a vector.
        for (k=1;k<=MAXDIM;k++) ix[k]=0;
```

```
        in=0;
        if (iv[1] != 1) return;
        fac=1.0/(1L << MAXBIT);
        for (j=1,k=0;j<=MAXBIT;j++,k+=MAXDIM) iu[j] = &iv[k];
        To allow both 1D and 2D addressing.
        for (k=1;k<=MAXDIM;k++) {
            for (j=1;j<=mdeg[k];j++) iu[j][k] <<= (MAXBIT-j);
            Stored values only require normalization.
            for (j=mdeg[k]+1;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=1;j<=MAXBIT;j++) {                  Find the rightmost zero bit.
            if (!(im & 1)) break;
            im >>= 1;
        }
        if (j > MAXBIT) nrerror("MAXBIT too small in sobseq");
        im=(j-1)*MAXDIM;
        for (k=1;k<=IMIN(*n,MAXDIM);k++) {         XOR the appropriate direction num-
            ix[k] ^= iv[im+k];                         ber into each component of the
            x[k]=ix[k]*fac;                            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, the minor radial coordinate r is defined by

$$r = \left([(x^2 + y^2)^{1/2} - R_0]^2 + z^2 \right)^{1/2} \tag{7.7.4}$$

Let us try the function

$$f(x,y,z) = \begin{cases} 1 + \cos\left(\dfrac{\pi r^2}{a^2}\right) & r < r_0 \\ 0 & r \geq r_0 \end{cases} \tag{7.7.5}$$

which can be integrated analytically in cylindrical coordinates, giving

$$\int\int\int dx\, dy\, dz\, f(x,y,z) = 2\pi^2 a^2 R_0 \tag{7.7.6}$$

With parameters $R_0 = 0.6$, $r_0 = 0.3$, we did 100 successive Monte Carlo integrations of equation (7.7.4), 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 7.7.2 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'

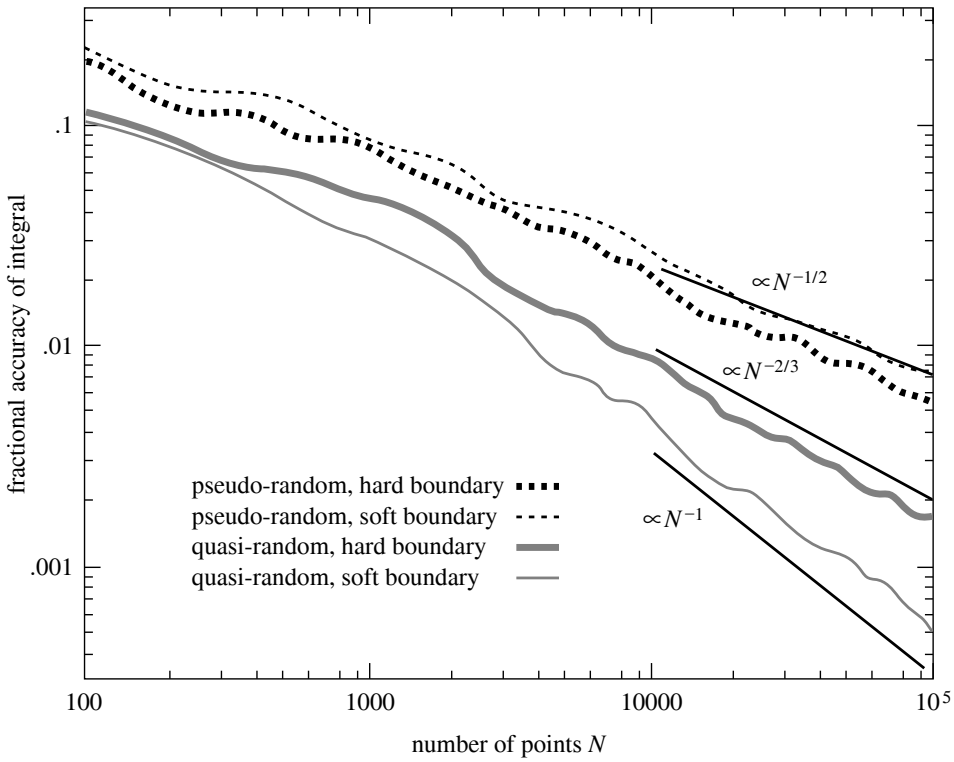

Figure 7.7.2. Fractional accuracy of Monte Carlo integrations as a function of number of points sampled, for two different integrands and two different methods of choosing random points. The quasi-random Sobol' sequence converges much more rapidly than a conventional pseudo-random sequence. Quasi-random 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. average of 100 trials.

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 7.7.2, 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 pseudorandom 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, zero outside,

$$f(x,y,z) = \begin{cases} 1 & r < r_0 \\ 0 & r \geq r_0 \end{cases} \tag{7.7.7}$$

where r is defined in equation (7.7.4). Not by coincidence, this function has the same analytic integral as the function of equation (7.7.5), namely $2\pi^2 a^2 R_0$.

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 7.7.2 by the thicker gray curve. The thicker dashed curve in Figure 7.7.2 is the result of integrating the function of equation (7.7.7) 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.

The Latin Hypercube

We might here give passing mention 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 4 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 4 parameters and 3 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, that parameter will be found with this sampling technique. On the other hand, if there is an important interaction among different design parameters, then the Latin hypercube gives no particular advantage. Use with care.

CITED REFERENCES AND FURTHER READING:

Halton, J.H. 1960, *Numerische Mathematik*, vol. 2, pp. 84–90. [1]

Bratley P., and Fox, B.L. 1988, *ACM Transactions on Mathematical Software*, vol. 14, pp. 88–100. [2]

Lambert, J.P. 1988, in *Numerical Mathematics – Singapore 1988*, ISNM vol. 86, R.P. Agarwal, Y.M. Chow, and S.J. Wilson, eds. (Basel: Birkhaüser), pp. 273–284.

Niederreiter, H. 1988, in *Numerical Integration III*, ISNM vol. 85, H. Brass and G. Hämmerlin, eds. (Basel: Birkhaüser), pp. 157–171.

Sobol', I.M. 1967, *USSR Computational Mathematics and Mathematical Physics*, vol. 7, no. 4, pp. 86–112. [3]

Antonov, I.A., and Saleev, V.M 1979, *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].

7.8 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` [3]. The techniques that we discuss all fall under the general rubric of *reduction of variance* (§7.6), but are otherwise quite distinct.

Importance Sampling

The use of *importance sampling* was already implicit in equations (7.6.6) and (7.6.7). 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

$$\int f\,dV = \int (f/g)\,g dV = \int h\,g dV \tag{7.8.1}$$

In equation (7.6.7) we interpreted equation (7.8.1) as suggesting a change of variable to G, the indefinite integral of g. That made gdV a perfect differential. We then proceeded to use the basic theorem of Monte Carlo integration, equation (7.6.1). A more general interpretation of equation (7.8.1) is that we can integrate f by instead sampling h — not, however, with uniform probability density dV, but rather with nonuniform density gdV. In this second interpretation, the first interpretation follows as the special case, where the *means* of generating the nonuniform sampling of gdV is via the transformation method, using the indefinite integral G (see §7.2).

More directly, one can go back and generalize the basic theorem (7.6.1) to the case of nonuniform sampling: Suppose that points x_i are chosen within the volume V with a probability density p satisfying

$$\int p\,dV = 1 \tag{7.8.2}$$

The generalized fundamental theorem is that the integral of any function f is estimated, using N sample points $x_1, \ldots, x_N$, by

$$I \equiv \int f\,dV = \int \frac{f}{p}\,pdV \approx \left\langle \frac{f}{p} \right\rangle \pm \sqrt{\frac{\langle f^2/p^2\rangle - \langle f/p\rangle^2}{N}} \tag{7.8.3}$$

where angle brackets denote arithmetic means over the N points, exactly as in equation (7.6.2). As in equation (7.6.1), the "plus-or-minus" term is a one standard deviation error estimate. Notice that equation (7.6.1) is in fact the special case of equation (7.8.3), 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 (7.8.3), which is the variance per sample point. Both angle brackets are themselves Monte Carlo estimators of integrals, so we can write

$$S \equiv \left\langle \frac{f^2}{p^2} \right\rangle - \left\langle \frac{f}{p} \right\rangle^2 \approx \int \frac{f^2}{p^2}\,pdV - \left[\int \frac{f}{p}\,pdV\right]^2 = \int \frac{f^2}{p}\,dV - \left[\int f\,dV\right]^2 \tag{7.8.4}$$

We now find the optimal p subject to the constraint equation (7.8.2) by the functional variation

$$0 = \frac{\delta}{\delta p}\left(\int \frac{f^2}{p}\,dV - \left[\int f\,dV\right]^2 + \lambda \int p\,dV\right) \tag{7.8.5}$$

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

$$p = \frac{|f|}{\sqrt{\lambda}} = \frac{|f|}{\int |f|\, dV} \tag{7.8.6}$$

where λ has been chosen to enforce the constraint (7.8.2).

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 7.8.4 and 7.8.6) is

$$S = S_{\text{optimal}} = \left(\int |f|\, dV \right)^2 - \left(\int f\, dV \right)^2 \tag{7.8.7}$$

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 §7.6) is that perfect knowledge of p in equation (7.8.6) 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 (7.8.7) is, but rather on how small is equation (7.8.4) for an *implementable* p, likely only a crude approximation to the ideal.

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:

$$\langle\langle f \rangle\rangle \equiv \frac{1}{V} \int f\, dV \qquad \langle f \rangle \equiv \frac{1}{N} \sum_i f(x_i) \tag{7.8.8}$$

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

$$\mathrm{Var}\,(\langle f \rangle) = \frac{\mathrm{Var}\,(f)}{N} \tag{7.8.9}$$

(compare equation 7.6.1).

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 (7.8.8), which we denote $\langle f \rangle'$, is

$$\langle f \rangle' \equiv \frac{1}{2} \left(\langle f \rangle_a + \langle f \rangle_b \right) \tag{7.8.10}$$

in other words, the mean of the sample averages in the two half-regions. The variance of estimator (7.8.10) is given by

$$\begin{aligned} \mathrm{Var}\left(\langle f \rangle'\right) &= \frac{1}{4} \left[\mathrm{Var}\left(\langle f \rangle_a\right) + \mathrm{Var}\left(\langle f \rangle_b\right) \right] \\ &= \frac{1}{4} \left[\frac{\mathrm{Var}_a\,(f)}{N/2} + \frac{\mathrm{Var}_b\,(f)}{N/2} \right] \\ &= \frac{1}{2N} \left[\mathrm{Var}_a\,(f) + \mathrm{Var}_b\,(f) \right] \end{aligned} \tag{7.8.11}$$

Here $\mathrm{Var}_a(f)$ 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

$$\mathrm{Var}(f) = \frac{1}{2}\left[\mathrm{Var}_a(f) + \mathrm{Var}_b(f)\right] + \frac{1}{4}\left(\langle\langle f\rangle\rangle_a - \langle\langle f\rangle\rangle_b\right)^2 \tag{7.8.12}$$

(In physics, this formula for combining second moments is the "parallel axis theorem.") Comparing equations (7.8.9), (7.8.11), and (7.8.12), 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

$$\mathrm{Var}\left(\langle f\rangle'\right) = \frac{1}{4}\left[\frac{\mathrm{Var}_a(f)}{N_a} + \frac{\mathrm{Var}_b(f)}{N - N_a}\right] \tag{7.8.13}$$

which is minimized (one can easily verify) when

$$\frac{N_a}{N} = \frac{\sigma_a}{\sigma_a + \sigma_b} \tag{7.8.14}$$

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 (7.8.14), then equation (7.8.13) reduces to

$$\mathrm{Var}\left(\langle f\rangle'\right) = \frac{(\sigma_a + \sigma_b)^2}{4N} \tag{7.8.15}$$

Equation (7.8.15) reduces to equation (7.8.9) 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.

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, that 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, §7.7, 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 (7.8.14) 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.

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*,

$$p \propto g(x, y, z, \ldots) = g_x(x) g_y(y) g_z(z) \ldots \tag{7.8.16}$$

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]

$$g_x(x) \propto \left[\int dy \int dz \ldots \frac{f^2(x, y, z, \ldots)}{g_y(y) g_z(z) \ldots} \right]^{1/2} \tag{7.8.17}$$

(and correspondingly for $y, z, \ldots$). Notice that this reduces to $g \propto |f|$ (7.8.6) in one dimension. Equation (7.8.17) 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 (7.8.17). 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

$$I_{\rm best} = \sum_{i=1}^{m} \frac{I_i}{\sigma_i^2} \Bigg/ \sum_{i=1}^{m} \frac{1}{\sigma_i^2} \qquad \sigma_{\rm best} = \left(\sum_{i=1}^{m} \frac{1}{\sigma_i^2} \right)^{-1/2} \tag{7.8.18}$$

Also returned is the quantity

$$\chi^2/m \equiv \frac{1}{m-1} \sum_{i=1}^{m} \frac{(I_i - I_{\rm best})^2}{\sigma_i^2} \tag{7.8.19}$$

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

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 5 iterations of a small number of samples, then to do a single high accuracy integration on the optimized grid.

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

$$I_g = \sum_i w_i g(\mathbf{x}) \tag{7.8.20}$$

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.

```
#include <stdio.h>
#include <math.h>
#include "nrutil.h"
#define ALPH 1.5
#define NDMX 50
#define MXDIM 10
#define TINY 1.0e-30

extern long idum;                          For random number initialization in main.

void vegas(float regn[], int ndim, float (*fxn)(float [], float), int init,
    unsigned long ncall, int itmx, int nprn, float *tgral, float *sd,
    float *chi2a)
```

Performs Monte Carlo integration of a user-supplied ndim-dimensional function fxn over a rectangular volume specified by regn[1..2*ndim], 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 below). 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.

```
{
    float ran2(long *idum);
```

```
void rebin(float rc, int nd, float r[], float xin[], float xi[]);
static int i,it,j,k,mds,nd,ndo,ng,npg,ia[MXDIM+1],kg[MXDIM+1];
static float calls,dv2g,dxg,f,f2,f2b,fb,rc,ti,tsi,wgt,xjac,xn,xnd,xo;
static float d[NDMX+1][MXDIM+1],di[NDMX+1][MXDIM+1],dt[MXDIM+1],
    dx[MXDIM+1], r[NDMX+1],x[MXDIM+1],xi[MXDIM+1][NDMX+1],xin[NDMX+1];
static double schi,si,swgt;
```

Best make everything static, allowing restarts.

```
if (init <= 0) {                    Normal entry. Enter here on a cold start.
    mds=ndo=1;                      Change to mds=0 to disable stratified sampling,
    for (j=1;j<=ndim;j++) xi[j][1]=1.0;        i.e., use importance sampling only.
}
if (init <= 1) si=swgt=schi=0.0;
```

Enter here to inherit the grid from a previous call, but not its answers.

```
if (init <= 2) {                    Enter here to inherit the previous grid and its
    nd=NDMX;                            answers.
    ng=1;
    if (mds) {                      Set up for stratification.
        ng=(int)pow(ncall/2.0+0.25,1.0/ndim);
        mds=1;
        if ((2*ng-NDMX) >= 0) {
            mds = -1;
            npg=ng/NDMX+1;
            nd=ng/npg;
            ng=npg*nd;
        }
    }
    for (k=1,i=1;i<=ndim;i++) k *= ng;
    npg=IMAX(ncall/k,2);
    calls=(float)npg * (float)k;
    dxg=1.0/ng;
    for (dv2g=1,i=1;i<=ndim;i++) dv2g *= dxg;
    dv2g=SQR(calls*dv2g)/npg/npg/(npg-1.0);
    xnd=nd;
    dxg *= xnd;
    xjac=1.0/calls;
    for (j=1;j<=ndim;j++) {
        dx[j]=regn[j+ndim]-regn[j];
        xjac *= dx[j];
    }
    if (nd != ndo) {                Do binning if necessary.
        for (i=1;i<=IMAX(nd,ndo);i++) r[i]=1.0;
        for (j=1;j<=ndim;j++) rebin(ndo/xnd,nd,r,xin,xi[j]);
        ndo=nd;
    }
    if (nprn >= 0) {
        printf("%s: ndim= %3d ncall= %8.0f\n",
            " Input parameters for vegas",ndim,calls);
        printf("%28s it=%5d itmx=%5d\n"," ",it,itmx);
        printf("%28s nprn=%3d ALPH=%5.2f\n"," ",nprn,ALPH);
        printf("%28s mds=%3d nd=%4d\n"," ",mds,nd);
        for (j=1;j<=ndim;j++) {
            printf("%30s xl[%2d]= %11.4g xu[%2d]= %11.4g\n",
                " ",j,regn[j],j,regn[j+ndim]);
        }
    }
}
for (it=1;it<=itmx;it++) {
```

Main iteration loop. Can enter here (init ≥ 3) to do an additional itmx iterations with all other parameters unchanged.

```
    ti=tsi=0.0;
    for (j=1;j<=ndim;j++) {
        kg[j]=1;
        for (i=1;i<=nd;i++) d[i][j]=di[i][j]=0.0;
```

```
}
for (;;) {
    fb=f2b=0.0;
    for (k=1;k<=npg;k++) {
        wgt=xjac;
        for (j=1;j<=ndim;j++) {
            xn=(kg[j]-ran2(&idum))*dxg+1.0;
            ia[j]=IMAX(IMIN((int)(xn),NDMX),1);
            if (ia[j] > 1) {
                xo=xi[j][ia[j]]-xi[j][ia[j]-1];
                rc=xi[j][ia[j]-1]+(xn-ia[j])*xo;
            } else {
                xo=xi[j][ia[j]];
                rc=(xn-ia[j])*xo;
            }
            x[j]=regn[j]+rc*dx[j];
            wgt *= xo*xnd;
        }
        f=wgt*(*fxn)(x,wgt);
        f2=f*f;
        fb += f;
        f2b += f2;
        for (j=1;j<=ndim;j++) {
            di[ia[j]][j] += f;
            if (mds >= 0) d[ia[j]][j] += f2;
        }
    }
    f2b=sqrt(f2b*npg);
    f2b=(f2b-fb)*(f2b+fb);
    if (f2b <= 0.0) f2b=TINY;
    ti += fb;
    tsi += f2b;
    if (mds < 0) {              Use stratified sampling.
        for (j=1;j<=ndim;j++) d[ia[j]][j] += f2b;
    }
    for (k=ndim;k>=1;k--) {
        kg[k] %= ng;
        if (++kg[k] != 1) break;
    }
    if (k < 1) break;
}
tsi *= dv2g;                    Compute final results for this iteration.
wgt=1.0/tsi;
si += wgt*ti;
schi += wgt*ti*ti;
swgt += wgt;
*tgral=si/swgt;
*chi2a=(schi-si*(*tgral))/(it-0.9999);
if (*chi2a < 0.0) *chi2a = 0.0;
*sd=sqrt(1.0/swgt);
tsi=sqrt(tsi);
if (nprn >= 0) {
    printf("%s %3d : integral = %14.7g +/- %9.2g\n",
        " iteration no.",it,ti,tsi);
    printf("%s integral =%14.7g+/-%9.2g chi**2/IT n = %9.2g\n",
        " all iterations: ",*tgral,*sd,*chi2a);
    if (nprn) {
        for (j=1;j<=ndim;j++) {
            printf(" DATA FOR axis %2d\n",j);
            printf("%6s%13s%11s%13s%11s%13s\n",
                "X","delta i","X","delta i","X","delta i");
            for (i=1+nprn/2;i<=nd;i += nprn+2) {
                printf("%8.5f%12.4g%12.5f%12.4g%12.5f%12.4g\n",
                    xi[j][i],di[i][j],xi[j][i+1],
```

```
                        di[i+1][j],xi[j][i+2],di[i+2][j]);
                }
            }
        }
    }
    for (j=1;j<=ndim;j++) {
        xo=d[1][j];
        xn=d[2][j];
        d[1][j]=(xo+xn)/2.0;
        dt[j]=d[1][j];
        for (i=2;i<nd;i++) {
            rc=xo+xn;
            xo=xn;
            xn=d[i+1][j];
            d[i][j] = (rc+xn)/3.0;
            dt[j] += d[i][j];
        }
        d[nd][j]=(xo+xn)/2.0;
        dt[j] += d[nd][j];
    }
    for (j=1;j<=ndim;j++) {
        rc=0.0;
        for (i=1;i<=nd;i++) {
            if (d[i][j] < TINY) d[i][j]=TINY;
            r[i]=pow((1.0-d[i][j]/dt[j])/
                (log(dt[j])-log(d[i][j])),ALPH);
            rc += r[i];
        }
        rebin(rc/xnd,nd,r,xin,xi[j]);
    }
  }
}
```

Refine the grid. Consult references to understand the subtlety of this procedure. The refinement is damped, to avoid rapid, destabilizing changes, and also compressed in range by the exponent ALPH.

```
void rebin(float rc, int nd, float r[], float xin[], float xi[])
```

Utility routine used by vegas, to rebin a vector of densities xi into new bins defined by a vector r.

```
{
    int i,k=0;
    float dr=0.0,xn=0.0,xo=0.0;

    for (i=1;i<nd;i++) {
        while (rc > dr)
            dr += r[++k];
        if (k > 1) xo=xi[k-1];
        xn=xi[k];
        dr -= rc;
        xin[i]=xn-(xn-xo)*dr/r[k];
    }
    for (i=1;i<nd;i++) xi[i]=xin[i];
    xi[nd]=1.0;
}
```

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* [3] 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 (7.8.10) and (7.8.13), 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 towards 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 Var (f) in each of the regions resulting from a possible bisection of R,

$$\begin{aligned} R_{ai} \equiv & (\mathbf{x}_a, \mathbf{x}_b - \frac{1}{2}\mathbf{e}_i \cdot (\mathbf{x}_b - \mathbf{x}_a)\mathbf{e}_i) \\ R_{bi} \equiv & (\mathbf{x}_a + \frac{1}{2}\mathbf{e}_i \cdot (\mathbf{x}_b - \mathbf{x}_a)\mathbf{e}_i, \mathbf{x}_b) \end{aligned} \tag{7.8.21}$$

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 (7.8.15), 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 (7.8.15) to choose i, we should do this allocation according to equation (7.8.14).

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 (7.8.21). (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 (7.8.10) and the first line of equation (7.8.11).

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 (7.8.13)–(7.8.15) and derive the equations that we actually use instead of these. The right-hand side of equation (7.8.13) applies the familiar scaling law of equation (7.8.9) 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 (7.8.9) to hold. Rather, we might substitute for equation (7.8.13) the relation,

$$\text{Var}\left(\langle f \rangle'\right) = \frac{1}{4}\left[\frac{\text{Var}_a\,(f)}{N_a^\alpha} + \frac{\text{Var}_b\,(f)}{(N - N_a)^\alpha}\right] \tag{7.8.22}$$

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

$$\frac{N_a}{N} = \frac{\text{Var}_a\,(f)^{1/(1+\alpha)}}{\text{Var}_a\,(f)^{1/(1+\alpha)} + \text{Var}_b\,(f)^{1/(1+\alpha)}} \tag{7.8.23}$$

and that its minimum value is

$$\text{Var}\left(\langle f \rangle'\right) \propto \left[\text{Var}_a\,(f)^{1/(1+\alpha)} + \text{Var}_b\,(f)^{1/(1+\alpha)}\right]^{1+\alpha} \tag{7.8.24}$$

Equations (7.8.22)–(7.8.24) reduce to equations (7.8.13)–(7.8.15) when $\alpha = 1$. Numerical experiments to find a self-consistent value for α find that $\alpha \approx 2$. That is, when equation (7.8.23) 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 (7.8.23) 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 (7.8.23) 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 (7.8.23) 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 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 (7.8.23) in all d directions. At 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)...` 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 (7.8.8), since the volume V of the rectangular region is known.

```
#include <stdlib.h>
#include <math.h>
#include "nrutil.h"
#define PFAC 0.1
#define MNPT 15
#define MNBS 60
#define TINY 1.0e-30
#define BIG 1.0e30
```

Here `PFAC` is the fraction of remaining function evaluations used *at each stage* to explore the variance of `func`. At least `MNPT` function evaluations are performed in any terminal subregion; a subregion is further bisected only if at least `MNBS` function evaluations are available. We take `MNBS` $= 4$*`MNPT`.

```
static long iran=0;

void miser(float (*func)(float []), float regn[], int ndim, unsigned long npts,
    float dith, float *ave, float *var)
```

Monte Carlo samples a user-supplied `ndim`-dimensional function `func` in a rectangular volume specified by `regn[1..2*ndim]`, 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`.

```
{
    void ranpt(float pt[], float regn[], int n);
    float *regn_temp;
```

```
unsigned long n,npre,nptl,nptr;
int j,jb;
float avel,varl;
float fracl,fval;
float rgl,rgm,rgr,s,sigl,siglb,sigr,sigrb;
float sum,sumb,summ,summ2;
float *fmaxl,*fmaxr,*fminl,*fminr;
float *pt,*rmid;

pt=vector(1,ndim);
if (npts < MNBS) {                          Too few points to bisect; do straight
    summ=summ2=0.0;                             Monte Carlo.
    for (n=1;n<=npts;n++) {
        ranpt(pt,regn,ndim);
        fval=(*func)(pt);
        summ += fval;
        summ2 += fval * fval;
    }
    *ave=summ/npts;
    *var=FMAX(TINY,(summ2-summ*summ/npts)/(npts*npts));
}
else {                                      Do the preliminary (uniform) sampling.
    rmid=vector(1,ndim);
    npre=LMAX((unsigned long)(npts*PFAC),MNPT);
    fmaxl=vector(1,ndim);
    fmaxr=vector(1,ndim);
    fminl=vector(1,ndim);
    fminr=vector(1,ndim);
    for (j=1;j<=ndim;j++) {                 Initialize the left and right bounds for
        iran=(iran*2661+36979) % 175000;        each dimension.
        s=SIGN(dith,(float)(iran-87500));
        rmid[j]=(0.5+s)*regn[j]+(0.5-s)*regn[ndim+j];
        fminl[j]=fminr[j]=BIG;
        fmaxl[j]=fmaxr[j] = -BIG;
    }
    for (n=1;n<=npre;n++) {                 Loop over the points in the sample.
        ranpt(pt,regn,ndim);
        fval=(*func)(pt);
        for (j=1;j<=ndim;j++) {             Find the left and right bounds for each
            if (pt[j]<=rmid[j]) {               dimension.
                fminl[j]=FMIN(fminl[j],fval);
                fmaxl[j]=FMAX(fmaxl[j],fval);
            }
            else {
                fminr[j]=FMIN(fminr[j],fval);
                fmaxr[j]=FMAX(fmaxr[j],fval);
            }
        }
    }
    sumb=BIG;                               Choose which dimension jb to bisect.
    jb=0;
    siglb=sigrb=1.0;
    for (j=1;j<=ndim;j++) {
        if (fmaxl[j] > fminl[j] && fmaxr[j] > fminr[j]) {
            sigl=FMAX(TINY,pow(fmaxl[j]-fminl[j],2.0/3.0));
            sigr=FMAX(TINY,pow(fmaxr[j]-fminr[j],2.0/3.0));
            sum=sigl+sigr;                  Equation (7.8.24), see text.
            if (sum<=sumb) {
                sumb=sum;
                jb=j;
                siglb=sigl;
                sigrb=sigr;
            }
        }
```

```
        }
        free_vector(fminr,1,ndim);
        free_vector(fminl,1,ndim);
        free_vector(fmaxr,1,ndim);
        free_vector(fmaxl,1,ndim);
        if (!jb) jb=1+(ndim*iran)/175000;      MNPT may be too small.
        rgl=regn[jb];                          Apportion the remaining points between
        rgm=rmid[jb];                              left and right.
        rgr=regn[ndim+jb];
        fracl=fabs((rgm-rgl)/(rgr-rgl));
        nptl=(unsigned long)(MNPT+(npts-npre-2*MNPT)*fracl*siglb
            /(fracl*siglb+(1.0-fracl)*sigrb));              Equation (7.8.23).
        nptr=npts-npre-nptl;
        regn_temp=vector(1,2*ndim);            Now allocate and integrate the two sub-
        for (j=1;j<=ndim;j++) {                    regions.
            regn_temp[j]=regn[j];
            regn_temp[ndim+j]=regn[ndim+j];
        }
        regn_temp[ndim+jb]=rmid[jb];
        miser(func,regn_temp,ndim,nptl,dith,&avel,&varl);
        regn_temp[jb]=rmid[jb];                Dispatch recursive call; will return back
        regn_temp[ndim+jb]=regn[ndim+jb];          here eventually.
        miser(func,regn_temp,ndim,nptr,dith,ave,var);
        free_vector(regn_temp,1,2*ndim);
        *ave=fracl*avel+(1-fracl)*(*ave);
        *var=fracl*fracl*varl+(1-fracl)*(1-fracl)*(*var);
        Combine left and right regions by equation (7.8.11) (1st line).
        free_vector(rmid,1,ndim);
    }
    free_vector(pt,1,ndim);
}
```

The `miser` routine calls a short function `ranpt` to get a random point within a specified d-dimensional region. The following version of `ranpt` 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` (§7.7). 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.)

```
extern long idum;

void ranpt(float pt[], float regn[], int n)
Returns a uniformly random point pt in an n-dimensional rectangular region. Used by miser;
calls ran1 for uniform deviates. Your main program should initialize the global variable idum
to a negative seed integer.
{
    float ran1(long *idum);
    int j;

    for (j=1;j<=n;j++)
        pt[j]=regn[j]+(regn[n+j]-regn[j])*ran1(&idum);
}
```

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* (New York: Springer-Verlag).

Lepage, G.P. 1978, *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]

Press, W.H., and Farrar, G.R. 1990, *Computers in Physics*, vol. 4, pp. 190–195. [3]