```
        } 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

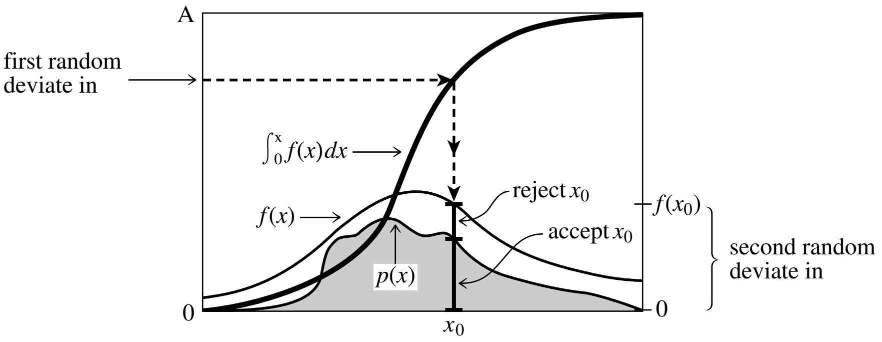

Figure 7.3.1. Rejection method for generating a random deviate x from a known probability distribution $p(x)$ that is everywhere less than some other function $f(x)$. The transformation method is first used to generate a random deviate x of the distribution f (compare Figure 7.2.1). A second uniform 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 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 $a_0 c_0$ (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 {                                    These four lines generate the tan-
                    v1=ran1(idum);                          gent of a random angle, i.e., they
                    v2=2.0*ran1(idum)-1.0;                  are equivalent to
                } while (v1*v1+v2*v2 > 1.0);                y = tan(π * ran1(idum)).
                y=v2/v1;
                am=ia-1;
                s=sqrt(2.0*am+1.0);
                x=s*y+am;                               We decide whether to reject x:
            } while (x <= 0.0);                         Reject in region of zero probability.
            e=(1.0+y*y)*exp(am*log(x/am)-s*y);          Ratio of prob. fn. to comparison fn.
        } while (ran1(idum) > e);                       Reject on basis of a second uniform
    }                                                       deviate.
    return x;
}
```


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:

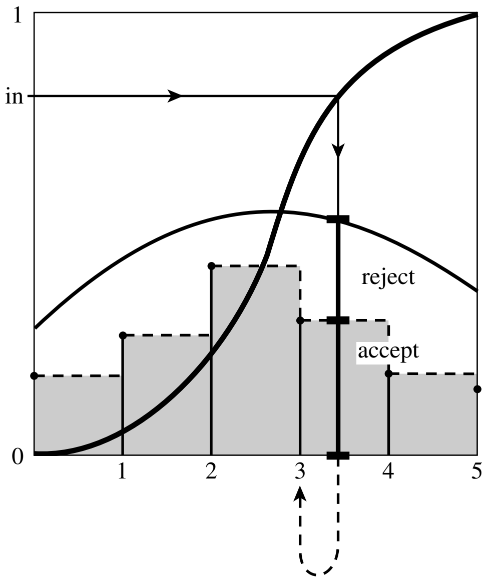

Figure 7.3.2. Rejection method as applied to an integer-valued distribution. The method is performed on the step function shown as a dashed line, yielding a real-valued deviate. This deviate is rounded down to the next lower integer, which is output.

```
#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,