CHAPTER 6

Special Functions

6.0 Introduction

There is nothing particularly special about a *special function*, except that some person in authority or a textbook writer (not the same thing!) has decided to bestow the moniker. Special functions are sometimes called *higher transcendental functions* (higher than what?) or *functions of mathematical physics* (but they occur in other fields also) or *functions that satisfy certain frequently occurring second-order differential equations* (but not all special functions do). One might simply call them "useful functions" and let it go at that. The choice of which functions to include in this chapter is highly arbitrary.

Commercially available program libraries contain many special function routines that are intended for users who will have no idea what goes on inside them. Such state-of-the-art black boxes are often very messy things, full of branches to completely different methods depending on the value of the calling arguments. Black boxes have, or should have, careful control of accuracy, to some stated uniform precision in all regimes.

We will not be quite so fastidious in our examples, in part because we want to illustrate techniques from Chapter 5, and in part because we *want* you to understand what goes on in the routines presented. Some of our routines have an accuracy parameter that can be made as small as desired, while others (especially those involving polynomial fits) give only a certain stated accuracy, one that we believe is serviceable (usually, but not always, close to double precision). We do *not* certify that the routines are perfect black boxes. We do hope that, if you ever encounter trouble in a routine, you will be able to diagnose and correct the problem on the basis of the information that we have given.

In short, the special function routines of this chapter are meant to be used — we use them all the time — but we also want you to learn from their inner workings.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at `http://www.nr.com/aands`.

Andrews, G.E., Askey, R., and Roy, R. 1999, *Special Functions* (Cambridge, UK: Cambridge University Press).

Thompson, W.J. 1997, *Atlas for Computing Mathematical Functions* (New York: Wiley-Interscience).

Spanier, J., and Oldham, K.B. 1987, *An Atlas of Functions* (Washington: Hemisphere Pub. Corp.).

Wolfram, S. 2003, *The Mathematica Book*, 5th ed. (Champaign, IL: Wolfram Media).

Hart, J.F., et al. 1968, *Computer Approximations* (New York: Wiley).

Hastings, C. 1955, *Approximations for Digital Computers* (Princeton: Princeton University Press).

Luke, Y.L. 1975, *Mathematical Functions and Their Approximations* (New York: Academic Press).

6.1 Gamma Function, Beta Function, Factorials, Binomial Coefficients

The gamma function is defined by the integral

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t}\, dt \tag{6.1.1}$$

When the argument z is an integer, the gamma function is just the familiar factorial function, but offset by 1:

$$n! = \Gamma(n+1) \tag{6.1.2}$$

The gamma function satisfies the recurrence relation

$$\Gamma(z+1) = z\,\Gamma(z) \tag{6.1.3}$$

If the function is known for arguments $z > 1$ or, more generally, in the half complex plane $\mathrm{Re}(z) > 1$, it can be obtained for $z < 1$ or $\mathrm{Re}\,(z) < 1$ by the reflection formula

$$\Gamma(1-z) = \frac{\pi}{\Gamma(z)\sin(\pi z)} = \frac{\pi z}{\Gamma(1+z)\sin(\pi z)} \tag{6.1.4}$$

Notice that $\Gamma(z)$ has a pole at $z = 0$ and at all negative integer values of z.

There are a variety of methods in use for calculating the function $\Gamma(z)$ numerically, but none is quite as neat as the approximation derived by Lanczos [1]. This scheme is entirely specific to the gamma function, seemingly plucked from thin air. We will not attempt to derive the approximation, but only state the resulting formula: For certain choices of rational γ and integer N, and for certain coefficients $c_1, c_2, \ldots, c_N$, the gamma function is given by

$$\Gamma(z+1) = (z+\gamma+\tfrac{1}{2})^{z+\frac{1}{2}} e^{-(z+\gamma+\frac{1}{2})} \times \sqrt{2\pi}\left[c_0 + \frac{c_1}{z+1} + \frac{c_2}{z+2} + \cdots + \frac{c_N}{z+N} + \epsilon\right] \quad (z > 0) \tag{6.1.5}$$

You can see that this is a sort of take-off on Stirling's approximation, but with a series of corrections that take into account the first few poles in the left complex plane. The constant c_0 is very nearly equal to 1. The error term is parametrized by

ϵ. For $N = 14$, and a certain set of c's and γ (calculated by P. Godfrey), the error is smaller than $|\epsilon| < 10^{-15}$. Even more impressive is the fact that, with these same constants, the formula (6.1.5) applies for the *complex* gamma function, *everywhere in the half complex plane* $\mathrm{Re}\, z > 0$, achieving almost the same accuracy as on the real line.

It is better to implement $\ln\Gamma(x)$ than $\Gamma(x)$, since the latter will overflow at quite modest values of x. Often the gamma function is used in calculations where the large values of $\Gamma(x)$ are divided by other large numbers, with the result being a perfectly ordinary value. Such operations would normally be coded as subtraction of logarithms. With (6.1.5) in hand, we can compute the logarithm of the gamma function with two calls to a logarithm and a few dozen arithmetic operations. This makes it not much more difficult than other built-in functions that we take for granted, such as $\sin x$ or e^x:

```
Doub gammln(const Doub xx) {
Returns the value ln[Γ(xx)] for xx > 0.
    Int j;
    Doub x,tmp,y,ser;
    static const Doub cof[14]={57.1562356658629235,-59.5979603554754912,
    14.1360979747417471,-0.491913816097620199,.339946499848118887e-4,
    .465236289270485756e-4,-.983744753048795646e-4,.158088703224912494e-3,
    -.210264441724104883e-3,.217439618115212643e-3,-.164318106536763890e-3,
    .844182239838527433e-4,-.261908384015814087e-4,.368991826595316234e-5};
    if (xx <= 0) throw("bad arg in gammln");
    y=x=xx;
    tmp = x+5.24218750000000000;          Rational 671/128.
    tmp = (x+0.5)*log(tmp)-tmp;
    ser = 0.999999999999997092;
    for (j=0;j<14;j++) ser += cof[j]/++y;
    return tmp+log(2.5066282746310005*ser/x);
}
```

4 gamma.h

How shall we write a routine for the factorial function $n!$? Generally the factorial function will be called for small integer values, and in most applications the same integer value will be called for many times. It is obviously inefficient to call `exp(gammln(n+1.))` for each required factorial. Better is to initialize a static table on the first call, and do a fast lookup on subsequent calls. The fixed size 171 for the table is because 170! is representable as an IEEE double precision value, but 171! overflows. It is also sometimes useful to know that factorials up to 22! have exact double precision representations (52 bits of mantissa, not counting powers of two that are absorbed into the exponent), while 23! and above are represented only approximately.

```
Doub factrl(const Int n) {
Returns the value n! as a floating-point number.
    static VecDoub a(171);
    static Bool init=true;
    if (init) {
        init = false;
        a[0] = 1.;
        for (Int i=1;i<171;i++) a[i] = i*a[i-1];
    }
    if (n < 0 || n > 170) throw("factrl out of range");
    return a[n];
}
```

5 gamma.h 6

More useful in practice is a function returning the log of a factorial, which doesn't have overflow issues. The size of the table of logarithms is whatever you can afford in space and initialization time. The value NTOP = 2000 should be increased if your integer arguments are often larger.

gamma.h

```
Doub factln(const Int n) {
Returns ln(n!).
    static const Int NTOP=2000;
    static VecDoub a(NTOP);
    static Bool init=true;
    if (init) {
        init = false;
        for (Int i=0;i<NTOP;i++) a[i] = gammln(i+1.);
    }
    if (n < 0) throw("negative arg in factln");
    if (n < NTOP) return a[n];
    return gammln(n+1.);                    Out of range of table.
}
```

The binomial coefficient is defined by

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \qquad 0 \le k \le n \tag{6.1.6}$$

A routine that takes advantage of the tables stored in `factrl` and `factln` is

gamma.h

```
Doub bico(const Int n, const Int k) {
Returns the binomial coefficient (n k) as a floating-point number.
    if (n<0 || k<0 || k>n) throw("bad args in bico");
    if (n<171) return floor(0.5+factrl(n)/(factrl(k)*factrl(n-k)));
    return floor(0.5+exp(factln(n)-factln(k)-factln(n-k)));
    The floor function cleans up roundoff error for smaller values of n and k.
}
```

If your problem requires a series of related binomial coefficients, a good idea is to use recurrence relations, for example,

$$\binom{n+1}{k} = \frac{n+1}{n-k+1}\binom{n}{k} = \binom{n}{k} + \binom{n}{k-1}$$

$$\binom{n}{k+1} = \frac{n-k}{k+1}\binom{n}{k} \tag{6.1.7}$$

Finally, turning away from the combinatorial functions with integer-valued arguments, we come to the beta function,

$$B(z,w) = B(w,z) = \int_0^1 t^{z-1}(1-t)^{w-1}\,dt \tag{6.1.8}$$

which is related to the gamma function by

$$B(z,w) = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)} \tag{6.1.9}$$

hence

```
Doub beta(const Doub z, const Doub w) {
Returns the value of the beta function B(z, w).
    return exp(gammln(z)+gammln(w)-gammln(z+w));
}
```

gamma.h

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at `http://www.nr.com/aands`, Chapter 6.

Lanczos, C. 1964, "A Precision Approximation of the Gamma Function," *SIAM Journal on Numerical Analysis*, ser. B, vol. 1, pp. 86–96.[1]

6.2 Incomplete Gamma Function and Error Function

The incomplete gamma function is defined by

$$P(a,x) \equiv \frac{\gamma(a,x)}{\Gamma(a)} \equiv \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt \qquad (a > 0) \tag{6.2.1}$$

It has the limiting values

$$P(a,0) = 0 \qquad \text{and} \qquad P(a,\infty) = 1 \tag{6.2.2}$$

The incomplete gamma function $P(a,x)$ is monotonic and (for a greater than one or so) rises from "near-zero" to "near-unity" in a range of x centered on about $a-1$, and of width about $\sqrt{a}$ (see Figure 6.2.1).

The complement of $P(a,x)$ is also confusingly called an incomplete gamma function,

$$Q(a,x) \equiv 1 - P(a,x) \equiv \frac{\Gamma(a,x)}{\Gamma(a)} \equiv \frac{1}{\Gamma(a)} \int_x^\infty e^{-t} t^{a-1} dt \qquad (a > 0) \tag{6.2.3}$$

It has the limiting values

$$Q(a,0) = 1 \qquad \text{and} \qquad Q(a,\infty) = 0 \tag{6.2.4}$$

The notations $P(a,x)$, $\gamma(a,x)$, and $\Gamma(a,x)$ are standard; the notation $Q(a,x)$ is specific to this book.

There is a series development for $\gamma(a,x)$ as follows:

$$\gamma(a,x) = e^{-x} x^a \sum_{n=0}^{\infty} \frac{\Gamma(a)}{\Gamma(a+1+n)} x^n \tag{6.2.5}$$

One does not actually need to compute a new $\Gamma(a+1+n)$ for each n; one rather uses equation (6.1.3) and the previous coefficient.

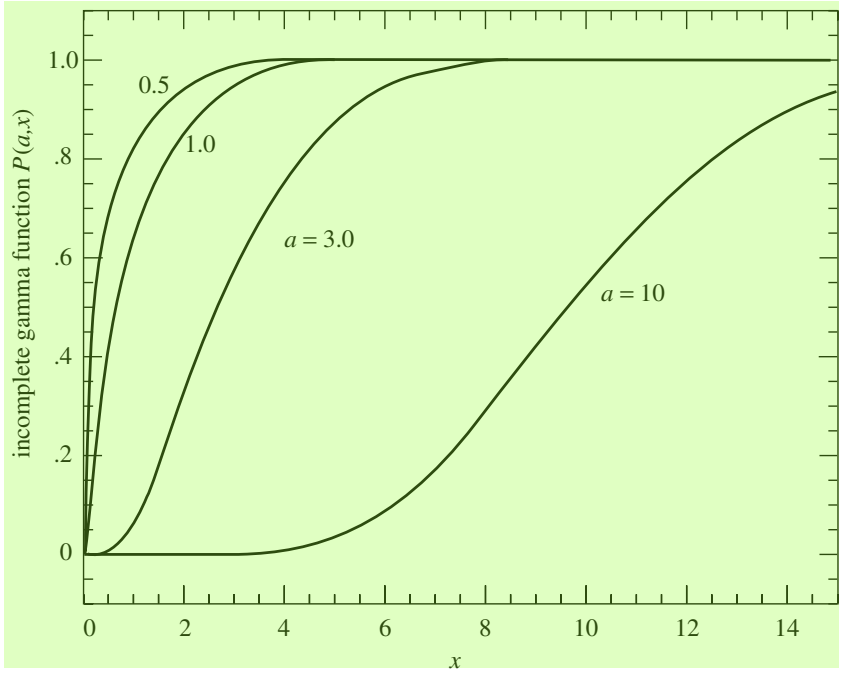

Figure 6.2.1. The incomplete gamma function $P(a,x)$ for four values of a.

A continued fraction development for $\Gamma(a,x)$ is

$$\Gamma(a,x) = e^{-x}x^a\left(\frac{1}{x+}\,\frac{1-a}{1+}\,\frac{1}{x+}\,\frac{2-a}{1+}\,\frac{2}{x+}\cdots\right) \qquad (x>0) \tag{6.2.6}$$

It is computationally better to use the even part of (6.2.6), which converges twice as fast (see §5.2):

$$\Gamma(a,x) = e^{-x}x^a\left(\frac{1}{x+1-a-}\,\frac{1\cdot(1-a)}{x+3-a-}\,\frac{2\cdot(2-a)}{x+5-a-}\cdots\right) \qquad (x>0) \tag{6.2.7}$$

It turns out that (6.2.5) converges rapidly for x less than about $a+1$, while (6.2.6) or (6.2.7) converges rapidly for x greater than about $a+1$. In these respective regimes each requires at most a few times $\sqrt{a}$ terms to converge, and this many only near $x=a$, where the incomplete gamma functions are varying most rapidly. For moderate values of a, less than 100, say, (6.2.5) and (6.2.7) together allow evaluation of the function for all x. An extra dividend is that we never need to compute a function value near zero by subtracting two nearly equal numbers.

Some applications require $P(a,x)$ and $Q(a,x)$ for much larger values of a, where both the series and the continued fraction are inefficient. In this regime, however, the integrand in equation (6.2.1) falls off sharply in both directions from its peak, within a few times $\sqrt{a}$. An efficient procedure is to evaluate the integral directly, with a single step of high-order Gauss-Legendre quadrature (§4.6) extending from x just far enough into the nearest tail to achieve negligible values of the integrand. Actually it is "half a step," because we need the dense abscissas only near x, not far out on the tail where the integrand is effectively zero.

We package the various incomplete gamma parts into an object `Gamma`. The only persistent state is the value `gln`, which is set to $\Gamma(a)$ for the most recent call to $P(a,x)$ or $Q(a,x)$. This is useful when you need a different normalization convention, for example $\gamma(a,x)$ or $\Gamma(a,x)$ in equations (6.2.1) or (6.2.3).

incgammabeta.h

```
struct Gamma : Gauleg18 {
```
Object for incomplete gamma function. `Gauleg18` provides coefficients for Gauss-Legendre quadrature.
```
    static const Int ASWITCH=100;                When to switch to quadrature method.
    static const Doub EPS;                       See end of struct for initializations.
    static const Doub FPMIN;
    Doub gln;

    Doub gammp(const Doub a, const Doub x) {
```
Returns the incomplete gamma function $P(a,x)$.
```
        if (x < 0.0 || a <= 0.0) throw("bad args in gammp");
        if (x == 0.0) return 0.0;
        else if ((Int)a >= ASWITCH) return gammpapprox(a,x,1); Quadrature.
        else if (x < a+1.0) return gser(a,x);        Use the series representation.
        else return 1.0-gcf(a,x);                    Use the continued fraction representation.
    }

    Doub gammq(const Doub a, const Doub x) {
```
Returns the incomplete gamma function $Q(a,x) \equiv 1 - P(a,x)$.
```
        if (x < 0.0 || a <= 0.0) throw("bad args in gammq");
        if (x == 0.0) return 1.0;
        else if ((Int)a >= ASWITCH) return gammpapprox(a,x,0); Quadrature.
        else if (x < a+1.0) return 1.0-gser(a,x);    Use the series representation.
        else return gcf(a,x);                        Use the continued fraction representation.
    }

    Doub gser(const Doub a, const Doub x) {
```
Returns the incomplete gamma function $P(a,x)$ evaluated by its series representation. Also sets $\ln\Gamma(a)$ as `gln`. User should not call directly.
```
        Doub sum,del,ap;
        gln=gammln(a);
        ap=a;
        del=sum=1.0/a;
        for (;;) {
            ++ap;
            del *= x/ap;
            sum += del;
            if (fabs(del) < fabs(sum)*EPS) {
                return sum*exp(-x+a*log(x)-gln);
            }
        }
    }

    Doub gcf(const Doub a, const Doub x) {
```
Returns the incomplete gamma function $Q(a,x)$ evaluated by its continued fraction representation. Also sets $\ln\Gamma(a)$ as `gln`. User should not call directly.
```
        Int i;
        Doub an,b,c,d,del,h;
        gln=gammln(a);
        b=x+1.0-a;                                   Set up for evaluating continued fraction
        c=1.0/FPMIN;                                     by modified Lentz's method (§5.2)
        d=1.0/b;                                         with $b_0 = 0$.
        h=d;
        for (i=1;;i++) {                             Iterate to convergence.
            an = -i*(i-a);
            b += 2.0;
            d=an*d+b;
```

```
            if (fabs(d) < FPMIN) d=FPMIN;
            c=b+an/c;
            if (fabs(c) < FPMIN) c=FPMIN;
            d=1.0/d;
            del=d*c;
            h *= del;
            if (fabs(del-1.0) <= EPS) break;
        }
        return exp(-x+a*log(x)-gln)*h;          Put factors in front.
    }

    Doub gammpapprox(Doub a, Doub x, Int psig) {
    Incomplete gamma by quadrature. Returns P(a,x) or Q(a,x), when psig is 1 or 0,
    respectively. User should not call directly.
        Int j;
        Doub xu,t,sum,ans;
        Doub a1 = a-1.0, lna1 = log(a1), sqrta1 = sqrt(a1);
        gln = gammln(a);
        Set how far to integrate into the tail:
        if (x > a1) xu = MAX(a1 + 11.5*sqrta1, x + 6.0*sqrta1);
        else xu = MAX(0.,MIN(a1 - 7.5*sqrta1, x - 5.0*sqrta1));
        sum = 0;
        for (j=0;j<ngau;j++) {                          Gauss-Legendre.
            t = x + (xu-x)*y[j];
            sum += w[j]*exp(-(t-a1)+a1*(log(t)-lna1));
        }
        ans = sum*(xu-x)*exp(a1*(lna1-1.)-gln);
        return (psig?(ans>0.0? 1.0-ans:-ans):(ans>=0.0? ans:1.0+ans));
    }

    Doub invgammp(Doub p, Doub a);
    Inverse function on x of P(a,x). See §6.2.1.

};
const Doub Gamma::EPS = numeric_limits<Doub>::epsilon();
const Doub Gamma::FPMIN = numeric_limits<Doub>::min()/EPS;
```

Remember that since `Gamma` is an object, you have to declare an instance of it before you can use its member functions. We habitually write

```
Gamma gam;
```

as a global declaration, and then call `gam.gammp` or `gam.gammq` as needed. The structure `Gauleg18` just contains the abscissas and weights for the Gauss-Legendre quadrature.

ncgammabeta.h

```
struct Gauleg18 {
Abscissas and weights for Gauss-Legendre quadrature.
    static const Int ngau = 18;
    static const Doub y[18];
    static const Doub w[18];
};
const Doub Gauleg18::y[18] = {0.0021695375159141994,
0.011413521097787704,0.027972308950302116,0.051727015600492421,
0.082502225484340941, 0.12007019910960293,0.16415283300752470,
0.21442376986779355, 0.27051082840644336, 0.33199876341447887,
0.39843234186401943, 0.46931971407375483, 0.54413605556657973,
0.62232745288031077, 0.70331500465597174, 0.78649910768313447,
0.87126389619061517, 0.9569818015262914 2};
const Doub Gauleg18::w[18] = {0.0055657196642445571,
0.012915947284065419,0.020181515297735382,0.027298621498568734,
0.034213810770299537,0.040875750923643261,0.047235083490265582,
0.053244713977759692,0.058860144245324798,0.064039797355015485,
```

```
0.068745323835736408,0.072941885005653087,0.076598410645870640,
0.079687828912071670,0.082187266704339706,0.084078218979661945,
0.085346685739338721,0.085983275670394821};
```

6.2.1 Inverse Incomplete Gamma Function

In many statistical applications one needs the inverse of the incomplete gamma function, that is, the value x such that $P(a,x) = p$, for a given value $0 \le p \le 1$. Newton's method works well if we can devise a good-enough initial guess. In fact, this is a good place to use Halley's method (see §9.4), since the second derivative (that is, the first derivative of the integrand) is easy to compute.

For $a > 1$, we use an initial guess that derives from §26.2.22 and §26.4.17 in reference [1]. For $a \le 1$ we first roughly approximate $P_a \equiv P(a,1)$:

$$P_a \equiv P(a,1) \approx 0.253a + 0.12a^2, \qquad 0 \le a \le 1 \tag{6.2.8}$$

and then solve for x in one or the other of the (rough) approximations:

$$P(a,x) \approx \begin{cases} P_a x^a, & x < 1 \\ P_a + (1-P_a)(1-e^{1-x}), & x \ge 1 \end{cases} \tag{6.2.9}$$

An implementation is

incgammabeta.h

```
Doub Gamma::invgammp(Doub p, Doub a) {
Returns x such that P(a,x) = p for an argument p between 0 and 1.
    Int j;
    Doub x,err,t,u,pp,lna1,afac,a1=a-1;
    const Doub EPS=1.e-8;                          Accuracy is the square of EPS.
    gln=gammln(a);
    if (a <= 0.) throw("a must be pos in invgammap");
    if (p >= 1.) return MAX(100.,a + 100.*sqrt(a));
    if (p <= 0.) return 0.0;
    if (a > 1.) {                                  Initial guess based on reference [1].
        lna1=log(a1);
        afac = exp(a1*(lna1-1.)-gln);
        pp = (p < 0.5)? p : 1. - p;
        t = sqrt(-2.*log(pp));
        x = (2.30753+t*0.27061)/(1.+t*(0.99229+t*0.04481)) - t;
        if (p < 0.5) x = -x;
        x = MAX(1.e-3,a*pow(1.-1./(9.*a)-x/(3.*sqrt(a)),3));
    } else {                                       Initial guess based on equations (6.2.8)
        t = 1.0 - a*(0.253+a*0.12);                    and (6.2.9).
        if (p < t) x = pow(p/t,1./a);
        else x = 1.-log(1.-(p-t)/(1.-t));
    }
    for (j=0;j<12;j++) {
        if (x <= 0.0) return 0.0;                  x too small to compute accurately.
        err = gammp(a,x) - p;
        if (a > 1.) t = afac*exp(-(x-a1)+a1*(log(x)-lna1));
        else t = exp(-x+a1*log(x)-gln);
        u = err/t;
        x -= (t = u/(1.-0.5*MIN(1.,u*((a-1.)/x - 1))));          Halley's method.
        if (x <= 0.) x = 0.5*(x + t);      Halve old value if x tries to go negative.
        if (fabs(t) < EPS*x ) break;
    }
    return x;
}
```

6.2.2 Error Function

The error function and complementary error function are special cases of the incomplete gamma function and are obtained moderately efficiently by the above procedures. Their definitions are

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt \tag{6.2.10}$$

and

$$\mathrm{erfc}(x) \equiv 1 - \mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2}\, dt \tag{6.2.11}$$

The functions have the following limiting values and symmetries:

$$\mathrm{erf}(0) = 0 \qquad \mathrm{erf}(\infty) = 1 \qquad \mathrm{erf}(-x) = -\mathrm{erf}(x) \tag{6.2.12}$$

$$\mathrm{erfc}(0) = 1 \qquad \mathrm{erfc}(\infty) = 0 \qquad \mathrm{erfc}(-x) = 2 - \mathrm{erfc}(x) \tag{6.2.13}$$

They are related to the incomplete gamma functions by

$$\mathrm{erf}(x) = P\left(\frac{1}{2}, x^2\right) \qquad (x \geq 0) \tag{6.2.14}$$

and

$$\mathrm{erfc}(x) = Q\left(\frac{1}{2}, x^2\right) \qquad (x \geq 0) \tag{6.2.15}$$

A faster calculation takes advantage of an approximation of the form

$$\mathrm{erfc}(z) \approx t\, \exp[-z^2 + \mathcal{P}(t)], \qquad z > 0 \tag{6.2.16}$$

where

$$t \equiv \frac{2}{2+z} \tag{6.2.17}$$

and $\mathcal{P}(t)$ is a polynomial for $0 \leq t \leq 1$ that can be found by Chebyshev methods (§5.8). As with `Gamma`, implementation is by an object that also includes the inverse function, here an inverse for both erf and erfc. Halley's method is again used for the inverses (as suggested by P.J. Acklam).

erf.h

```
struct Erf {
Object for error function and related functions.
    static const Int ncof=28;
    static const Doub cof[28];                    Initialization at end of struct.

    inline Doub erf(Doub x) {
    Return erf(x) for any x.
        if (x >=0.) return 1.0 - erfccheb(x);
        else return erfccheb(-x) - 1.0;
    }

    inline Doub erfc(Doub x) {
    Return erfc(x) for any x.
        if (x >= 0.) return erfccheb(x);
        else return 2.0 - erfccheb(-x);
    }
```

Doub erfccheb(Doub z){
Evaluate equation (6.2.16) using stored Chebyshev coefficients. User should not call directly.

```
Int j;
Doub t,ty,tmp,d=0.,dd=0.;
if (z < 0.) throw("erfccheb requires nonnegative argument");
t = 2./(2.+z);
ty = 4.*t - 2.;
for (j=ncof-1;j>0;j--) {
    tmp = d;
    d = ty*d - dd + cof[j];
    dd = tmp;
}
return t*exp(-z*z + 0.5*(cof[0] + ty*d) - dd);
```

}

Doub inverfc(Doub p) {
Inverse of complementary error function. Returns x such that $\mathrm{erfc}(x) = p$ for argument p between 0 and 2.

```
Doub x,err,t,pp;
if (p >= 2.0) return -100.;          Return arbitrary large pos or neg value.
if (p <= 0.0) return 100.;
pp = (p < 1.0)? p : 2. - p;
t = sqrt(-2.*log(pp/2.));            Initial guess:
x = -0.70711*((2.30753+t*0.27061)/(1.+t*(0.99229+t*0.04481)) - t);
for (Int j=0;j<2;j++) {
    err = erfc(x) - pp;
    x += err/(1.1283791670955125 7*exp(-SQR(x))-x*err);  Halley.
}
return (p < 1.0? x : -x);
```

}

inline Doub inverf(Doub p) {return inverfc(1.-p);}
Inverse of the error function. Returns x such that $\mathrm{erf}(x) = p$ for argument p between -1 and 1.

};

```
const Doub Erf::cof[28] = {-1.3026537197817094, 6.4196979235649026e-1,
    1.9476473204185836e-2,-9.561514786808631e-3,-9.46595344482036e-4,
    3.66839497852761e-4,4.2523324806907e-5,-2.0278578112534e-5,
    -1.624290004647e-6,1.303655835580e-6,1.5626441722e-8,-8.5238095915e-8,
    6.529054439e-9,5.059343495e-9,-9.91364156e-10,-2.27365122e-10,
    9.6467911e-11, 2.394038e-12,-6.886027e-12,8.94487e-13, 3.13092e-13,
    -1.12708e-13,3.81e-16,7.106e-15,-1.523e-15,-9.4e-17,1.21e-16,-2.8e-17};
```

A lower-order Chebyshev approximation produces a very concise routine, though with only about single precision accuracy:

Doub erfcc(const Doub x) erf.h
Returns the complementary error function erfc(x) with fractional error everywhere less than 1.2×10^{-7}.

```
{
    Doub t,z=fabs(x),ans;
    t=2./(2.+z);
    ans=t*exp(-z*z-1.26551223+t*(1.00002368+t*(0.37409196+t*(0.09678418+
        t*(-0.18628806+t*(0.27886807+t*(-1.13520398+t*(1.48851587+
        t*(-0.82215223+t*0.17087277)))))))));
    return (x >= 0.0 ? ans : 2.0-ans);
}
```

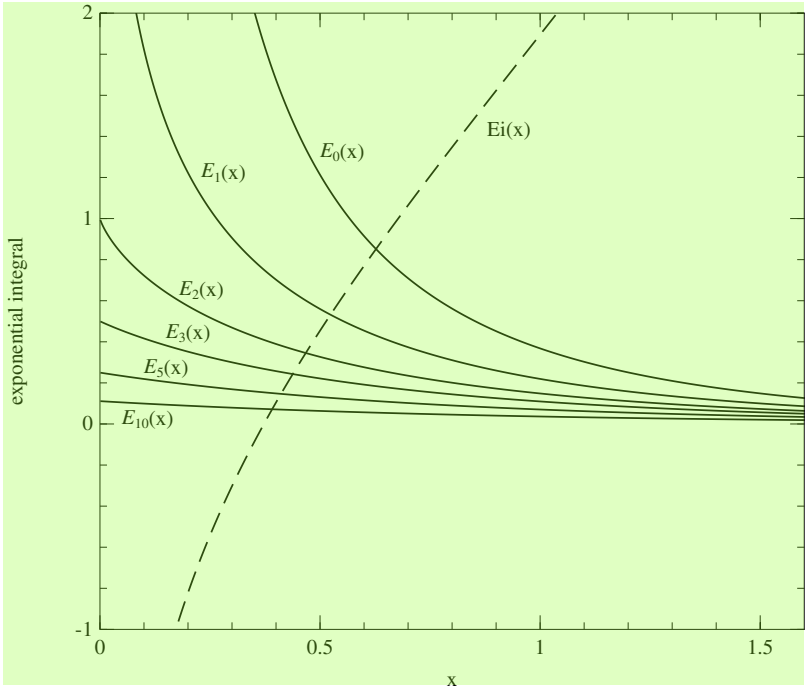

Figure 6.3.1. Exponential integrals $E_n(x)$ for $n = 0, 1, 2, 3, 5$, and 10, and the exponential integral $\mathrm{Ei}(x)$.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at `http://www.nr.com/aands`, Chapters 6, 7, and 26.[1]

Pearson, K. (ed.) 1951, *Tables of the Incomplete Gamma Function* (Cambridge, UK: Cambridge University Press).

6.3 Exponential Integrals

The standard definition of the exponential integral is

$$E_n(x) = \int_1^\infty \frac{e^{-xt}}{t^n}\,dt, \qquad x > 0, \quad n = 0, 1, \ldots \tag{6.3.1}$$

The function defined by the principal value of the integral

$$\mathrm{Ei}(x) = -\int_{-x}^{\infty} \frac{e^{-t}}{t}\,dt = \int_{-\infty}^{x} \frac{e^{t}}{t}\,dt, \qquad x > 0 \tag{6.3.2}$$

is also called an exponential integral. Note that $\mathrm{Ei}(-x)$ is related to $-E_1(x)$ by analytic continuation. Figure 6.3.1 plots these functions for representative values of their parameters.

The function $E_n(x)$ is a special case of the incomplete gamma function

$$E_n(x) = x^{n-1}\Gamma(1-n,x) \tag{6.3.3}$$

We can therefore use a similar strategy for evaluating it. The continued fraction — just equation (6.2.6) rewritten — converges for all $x > 0$:

$$E_n(x) = e^{-x}\left(\frac{1}{x+}\ \frac{n}{1+}\ \frac{1}{x+}\ \frac{n+1}{1+}\ \frac{2}{x+}\cdots\right) \tag{6.3.4}$$

We use it in its more rapidly converging even form,

$$E_n(x) = e^{-x}\left(\frac{1}{x+n-}\ \frac{1\cdot n}{x+n+2-}\ \frac{2(n+1)}{x+n+4-}\cdots\right) \tag{6.3.5}$$

The continued fraction only really converges fast enough to be useful for $x \gtrsim 1.5$. For $0 < x \lesssim 1$ we can use the series representation

$$E_n(x) = \frac{(-x)^{n-1}}{(n-1)!}[-\ln x + \psi(n)] - \sum_{\substack{m=0\\ m\neq n-1}}^{\infty} \frac{(-x)^m}{(m-n+1)m!} \tag{6.3.6}$$

The quantity $\psi(n)$ here is the digamma function, given for integer arguments by

$$\psi(1) = -\gamma, \qquad \psi(n) = -\gamma + \sum_{m=1}^{n-1}\frac{1}{m} \tag{6.3.7}$$

where $\gamma = 0.5772156649\ldots$ is Euler's constant. We evaluate the expression (6.3.6) in order of ascending powers of x:

$$\begin{aligned} E_n(x) = &-\left[\frac{1}{(1-n)} - \frac{x}{(2-n)\cdot 1} + \frac{x^2}{(3-n)(1\cdot 2)} - \cdots + \frac{(-x)^{n-2}}{(-1)(n-2)!}\right] \\ &+ \frac{(-x)^{n-1}}{(n-1)!}[-\ln x + \psi(n)] - \left[\frac{(-x)^n}{1\cdot n!} + \frac{(-x)^{n+1}}{2\cdot(n+1)!} + \cdots\right] \end{aligned} \tag{6.3.8}$$

The first square bracket is omitted when $n = 1$. This method of evaluation has the advantage that, for large n, the series converges before reaching the term containing $\psi(n)$. Accordingly, one needs an algorithm for evaluating $\psi(n)$ only for small n, $n \lesssim 20 - 40$. We use equation (6.3.7), although a table lookup would improve efficiency slightly.

Amos [1] presents a careful discussion of the truncation error in evaluating equation (6.3.8) and gives a fairly elaborate termination criterion. We have found that simply stopping when the last term added is smaller than the required tolerance works about as well.

Two special cases have to be handled separately:

$$\begin{aligned} E_0(x) &= \frac{e^{-x}}{x} \\ E_n(0) &= \frac{1}{n-1}, \qquad n > 1 \end{aligned} \tag{6.3.9}$$

The routine expint allows fast evaluation of $E_n(x)$ to any accuracy EPS within 1
the reach of your machine's precision for floating-point numbers. The only modification required for increased accuracy is to supply Euler's constant with enough significant digits. Wrench [2] can provide you with the first 328 digits if necessary!

expint.h

```
Doub expint(const Int n, const Doub x)                                    2
Evaluates the exponential integral E_n(x).
{
    static const Int MAXIT=100;
    static const Doub EULER=0.577215664901533,
        EPS=numeric_limits<Doub>::epsilon(),
        BIG=numeric_limits<Doub>::max()*EPS;
        Here MAXIT is the maximum allowed number of iterations; EULER is Euler's constant
        γ; EPS is the desired relative error, not smaller than the machine precision; BIG is a
        number near the largest representable floating-point number.
    Int i,ii,nm1=n-1;
    Doub a,b,c,d,del,fact,h,psi,ans;
    if (n < 0 || x < 0.0 || (x==0.0 && (n==0 || n==1)))
        throw("bad arguments in expint");
    if (n == 0) ans=exp(-x)/x;                          Special case.
    else {
        if (x == 0.0) ans=1.0/nm1;                      Another special case.
        else {
            if (x > 1.0) {                              Lentz's algorithm (§5.2).
                b=x+n;
                c=BIG;
                d=1.0/b;
                h=d;
                for (i=1;i<=MAXIT;i++) {
                    a = -i*(nm1+i);
                    b += 2.0;
                    d=1.0/(a*d+b);                      Denominators cannot be zero.
                    c=b+a/c;
                    del=c*d;
                    h *= del;
                    if (abs(del-1.0) <= EPS) {
                        ans=h*exp(-x);
                        return ans;
                    }
                }
                throw("continued fraction failed in expint");
            } else {                                    Evaluate series.
                ans = (nm1!=0 ? 1.0/nm1 : -log(x)-EULER);       Set first term.
                fact=1.0;
                for (i=1;i<=MAXIT;i++) {
                    fact *= -x/i;
                    if (i != nm1) del = -fact/(i-nm1);
                    else {
                        psi = -EULER;                   Compute ψ(n).
                        for (ii=1;ii<=nm1;ii++) psi += 1.0/ii;
                        del=fact*(-log(x)+psi);
                    }
                    ans += del;
                    if (abs(del) < abs(ans)*EPS) return ans;
                }
                throw("series failed in expint");
            }
        }
    }
    return ans;
}
```

A good algorithm for evaluating Ei is to use the power series for small x and the asymptotic series for large x. The power series is

$$\mathrm{Ei}(x) = \gamma + \ln x + \frac{x}{1\cdot 1!} + \frac{x^2}{2\cdot 2!} + \cdots \tag{6.3.10}$$

where γ is Euler's constant. The asymptotic expansion is

$$\mathrm{Ei}(x) \sim \frac{e^x}{x}\left(1 + \frac{1!}{x} + \frac{2!}{x^2} + \cdots\right) \tag{6.3.11}$$

The lower limit for the use of the asymptotic expansion is approximately $|\ln \mathrm{EPS}|$, where EPS is the required relative error.

expint.h

```
Doub ei(const Doub x) {
```
Computes the exponential integral $\mathrm{Ei}(x)$ for $x > 0$.
```
    static const Int MAXIT=100;
    static const Doub EULER=0.577215664901533,
        EPS=numeric_limits<Doub>::epsilon(),
        FPMIN=numeric_limits<Doub>::min()/EPS;
```
Here MAXIT is the maximum number of iterations allowed; EULER is Euler's constant γ; EPS is the relative error, or absolute error near the zero of Ei at $x = 0.3725$; FPMIN is a number close to the smallest representable floating-point number.
```
    Int k;
    Doub fact,prev,sum,term;
    if (x <= 0.0) throw("Bad argument in ei");
    if (x < FPMIN) return log(x)+EULER;
```
Special case: Avoid failure of convergence test because of underflow.
```
    if (x <= -log(EPS)) {
```
Use power series.
```
        sum=0.0;
        fact=1.0;
        for (k=1;k<=MAXIT;k++) {
            fact *= x/k;
            term=fact/k;
            sum += term;
            if (term < EPS*sum) break;
        }
        if (k > MAXIT) throw("Series failed in ei");
        return sum+log(x)+EULER;
    } else {
```
Use asymptotic series.
```
        sum=0.0;
```
Start with second term.
```
        term=1.0;
        for (k=1;k<=MAXIT;k++) {
            prev=term;
            term *= k/x;
            if (term < EPS) break;
```
Since final sum is greater than one, term itself approximates the relative error.
```
            if (term < prev) sum += term;
```
Still converging: Add new term.
```
            else {
                sum -= prev;
```
Diverging: Subtract previous term and exit.
```
                break;
            }
        }
        return exp(x)*(1.0+sum)/x;
    }
}
```

CITED REFERENCES AND FURTHER READING:

Stegun, I.A., and Zucker, R. 1974, "Automatic Computing Methods for Special Functions. II. The Exponential Integral $E_n(x)$," *Journal of Research of the National Bureau of Standards,*

vol. 78B, pp. 199–216; 1976, "Automatic Computing Methods for Special Functions. III. The Sine, Cosine, Exponential Integrals, and Related Functions," *op. cit.*, vol. 80B, pp. 291–311.

Amos D.E. 1980, "Computation of Exponential Integrals," *ACM Transactions on Mathematical Software*, vol. 6, pp. 365–377[1]; also vol. 6, pp. 420–428.

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at `http://www.nr.com/aands`, Chapter 5.

Wrench J.W. 1952, "A New Calculation of Euler's Constant," *Mathematical Tables and Other Aids to Computation*, vol. 6, p. 255.[2]

6.4 Incomplete Beta Function

The incomplete beta function is defined by

$$I_x(a,b) \equiv \frac{B_x(a,b)}{B(a,b)} \equiv \frac{1}{B(a,b)} \int_0^x t^{a-1}(1-t)^{b-1}\,dt \qquad (a,b>0) \tag{6.4.1}$$

It has the limiting values

$$I_0(a,b) = 0 \qquad I_1(a,b) = 1 \tag{6.4.2}$$

and the symmetry relation

$$I_x(a,b) = 1 - I_{1-x}(b,a) \tag{6.4.3}$$

If a and b are both rather greater than one, then $I_x(a,b)$ rises from "near-zero" to "near-unity" quite sharply at about $x = a/(a+b)$. Figure 6.4.1 plots the function for several pairs (a,b).

The incomplete beta function has a series expansion

$$I_x(a,b) = \frac{x^a(1-x)^b}{a\,B(a,b)} \left[1 + \sum_{n=0}^{\infty} \frac{B(a+1,n+1)}{B(a+b,n+1)} x^{n+1} \right] \tag{6.4.4}$$

but this does not prove to be very useful in its numerical evaluation. (Note, however, that the beta functions in the coefficients can be evaluated for each value of n with just the previous value and a few multiplies, using equations 6.1.9 and 6.1.3.)

The continued fraction representation proves to be much more useful:

$$I_x(a,b) = \frac{x^a(1-x)^b}{a\,B(a,b)} \left[\frac{1}{1+} \frac{d_1}{1+} \frac{d_2}{1+} \cdots \right] \tag{6.4.5}$$

where

$$\begin{aligned} d_{2m+1} &= -\frac{(a+m)(a+b+m)x}{(a+2m)(a+2m+1)} \\ d_{2m} &= \frac{m(b-m)x}{(a+2m-1)(a+2m)} \end{aligned} \tag{6.4.6}$$

This continued fraction converges rapidly for $x < (a+1)/(a+b+2)$, except when a and b are both large, when it can take $O(\sqrt{\min(a,b)})$ iterations. For $x >$

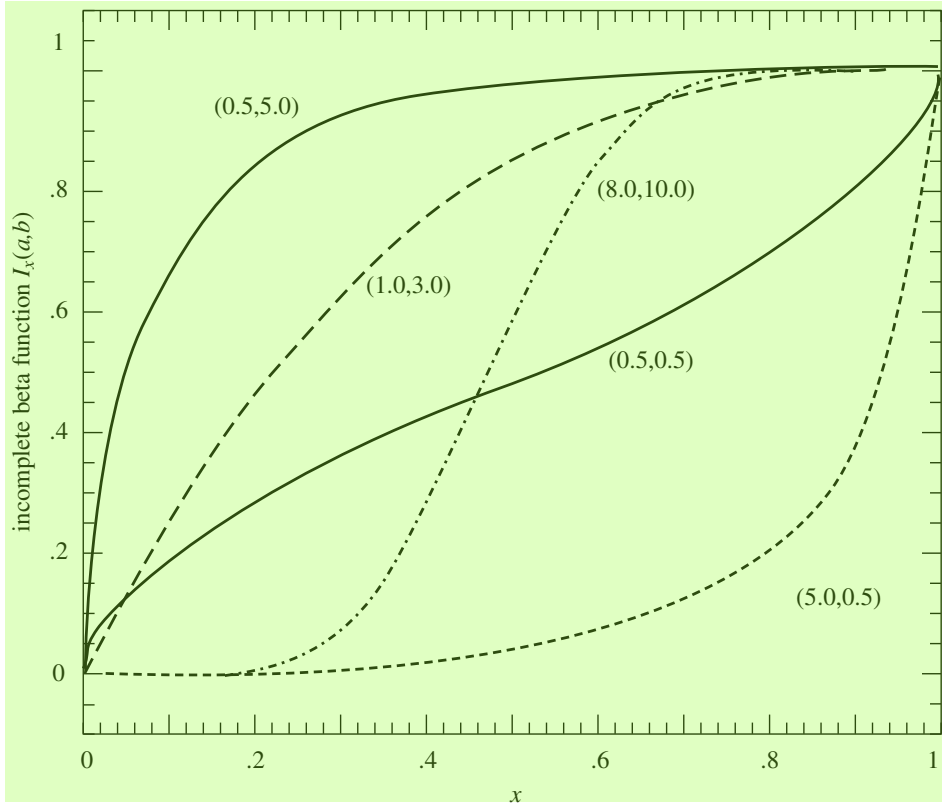

Figure 6.4.1. The incomplete beta function $I_x(a,b)$ for five different pairs of (a,b). Notice that the pairs $(0.5, 5.0)$ and $(5.0, 0.5)$ are symmetrically related as indicated in equation (6.4.3).

$(a+1)/(a+b+2)$ we can just use the symmetry relation (6.4.3) to obtain an equivalent computation in which the convergence is again rapid. Our computational strategy is thus very similar to that used in `Gamma`: We use the continued fraction except when a and b are both large, in which case we do a single step of high-order Gauss-Legendre quadrature.

Also as in `Gamma`, we code an inverse function using Halley's method. When a and b are both ≥ 1 the initial guess comes from §26.5.22 in reference [1]. When either is less than 1, the guess comes from first crudely approximating

$$\int_0^1 t^{a-1}(1-t)^{b-1}dt \approx \frac{1}{a}\left(\frac{a}{a+b}\right)^a + \frac{1}{b}\left(\frac{b}{a+b}\right)^b \equiv S \tag{6.4.7}$$

which comes from breaking the integral at $t = a/(a+b)$ and ignoring one factor in the integrand on each side of the break. We then write

$$I_x(a,b) \approx \begin{cases} x^a/(Sa) & x \leq a/(a+b) \\ (1-x)^b/(Sb) & x > a/(a+b) \end{cases} \tag{6.4.8}$$

and solve for x in the respective regimes. While crude, this is good enough to get well within the basin of convergence in all cases.

ncgammabeta.h

```
struct Beta : Gauleg18 {
```
Object for incomplete beta function. `Gauleg18` provides coefficients for Gauss-Legendre quadrature.
```
    static const Int SWITCH=3000;
    static const Doub EPS, FPMIN;
```
When to switch to quadrature method. See end of `struct` for initializations.
```
    Doub betai(const Doub a, const Doub b, const Doub x) {
```
Returns incomplete beta function $I_x(a,b)$ for positive a and b, and x between 0 and 1.
```
        Doub bt;
        if (a <= 0.0 || b <= 0.0) throw("Bad a or b in routine betai");
        if (x < 0.0 || x > 1.0) throw("Bad x in routine betai");
        if (x == 0.0 || x == 1.0) return x;
        if (a > SWITCH && b > SWITCH) return betaiapprox(a,b,x);
        bt=exp(gammln(a+b)-gammln(a)-gammln(b)+a*log(x)+b*log(1.0-x));
        if (x < (a+1.0)/(a+b+2.0)) return bt*betacf(a,b,x)/a;
        else return 1.0-bt*betacf(b,a,1.0-x)/b;
    }

    Doub betacf(const Doub a, const Doub b, const Doub x) {
```
Evaluates continued fraction for incomplete beta function by modified Lentz's method (§5.2). User should not call directly.
```
        Int m,m2;
        Doub aa,c,d,del,h,qab,qam,qap;
        qab=a+b;
        qap=a+1.0;
        qam=a-1.0;
        c=1.0;
        d=1.0-qab*x/qap;
        if (fabs(d) < FPMIN) d=FPMIN;
        d=1.0/d;
        h=d;
        for (m=1;m<10000;m++) {
            m2=2*m;
            aa=m*(b-m)*x/((qam+m2)*(a+m2));
            d=1.0+aa*d;
            if (fabs(d) < FPMIN) d=FPMIN;
            c=1.0+aa/c;
            if (fabs(c) < FPMIN) c=FPMIN;
            d=1.0/d;
            h *= d*c;
            aa = -(a+m)*(qab+m)*x/((a+m2)*(qap+m2));
            d=1.0+aa*d;
            if (fabs(d) < FPMIN) d=FPMIN;
            c=1.0+aa/c;
            if (fabs(c) < FPMIN) c=FPMIN;
            d=1.0/d;
            del=d*c;
            h *= del;
            if (fabs(del-1.0) <= EPS) break;
        }
        return h;
    }
```
These q's will be used in factors that occur in the coefficients (6.4.6).

First step of Lentz's method.

One step (the even one) of the recurrence.

Next step of the recurrence (the odd one).

Are we done?
```
    Doub betaiapprox(Doub a, Doub b, Doub x) {
```
Incomplete beta by quadrature. Returns $I_x(a,b)$. User should not call directly.
```
        Int j;
        Doub xu,t,sum,ans;
        Doub a1 = a-1.0, b1 = b-1.0, mu = a/(a+b);
        Doub lnmu=log(mu),lnmuc=log(1.-mu);
        t = sqrt(a*b/(SQR(a+b)*(a+b+1.0)));
        if (x > a/(a+b)) {
            if (x >= 1.0) return 1.0;
            xu = MIN(1.,MAX(mu + 10.*t, x + 5.0*t));
        } else {
```
Set how far to integrate into the tail:

```
            if (x <= 0.0) return 0.0;
            xu = MAX(0.,MIN(mu - 10.*t, x - 5.0*t));
        }
        sum = 0;
        for (j=0;j<18;j++) {                    Gauss-Legendre.
            t = x + (xu-x)*y[j];
            sum += w[j]*exp(a1*(log(t)-lnmu)+b1*(log(1-t)-lnmuc));
        }
        ans = sum*(xu-x)*exp(a1*lnmu-gammln(a)+b1*lnmuc-gammln(b)+gammln(a+b));
        return ans>0.0? 1.0-ans : -ans;
    }

    Doub invbetai(Doub p, Doub a, Doub b) {
    Inverse of incomplete beta function. Returns x such that I_x(a,b) = p for argument p
    between 0 and 1.
        const Doub EPS = 1.e-8;
        Doub pp,t,u,err,x,al,h,w,afac,a1=a-1.,b1=b-1.;
        Int j;
        if (p <= 0.) return 0.;
        else if (p >= 1.) return 1.;
        else if (a >= 1. && b >= 1.) {          Set initial guess. See text.
            pp = (p < 0.5)? p : 1. - p;
            t = sqrt(-2.*log(pp));
            x = (2.30753+t*0.27061)/(1.+t*(0.99229+t*0.04481)) - t;
            if (p < 0.5) x = -x;
            al = (SQR(x)-3.)/6.;
            h = 2./(1./(2.*a-1.)+1./(2.*b-1.));
            w = (x*sqrt(al+h)/h)-(1./(2.*b-1)-1./(2.*a-1.))*(al+5./6.-2./(3.*h));
            x = a/(a+b*exp(2.*w));
        } else {
            Doub lna = log(a/(a+b)), lnb = log(b/(a+b));
            t = exp(a*lna)/a;
            u = exp(b*lnb)/b;
            w = t + u;
            if (p < t/w) x = pow(a*w*p,1./a);
            else x = 1. - pow(b*w*(1.-p),1./b);
        }
        afac = -gammln(a)-gammln(b)+gammln(a+b);
        for (j=0;j<10;j++) {
            if (x == 0. || x == 1.) return x;       a or b too small for accurate calcu-
                                                        lation.
            err = betai(a,b,x) - p;
            t = exp(a1*log(x)+b1*log(1.-x) + afac);
            u = err/t;                              Halley:
            x -= (t = u/(1.-0.5*MIN(1.,u*(a1/x - b1/(1.-x)))));
            if (x <= 0.) x = 0.5*(x + t);           Bisect if x tries to go neg or > 1.
            if (x >= 1.) x = 0.5*(x + t + 1.);
            if (fabs(t) < EPS*x && j > 0) break;
        }
        return x;
    }

};
const Doub Beta::EPS = numeric_limits<Doub>::epsilon();
const Doub Beta::FPMIN = numeric_limits<Doub>::min()/EPS;
```

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at `http://www.nr.com/aands`, Chapters 6 and 26.[1]

Pearson, E., and Johnson, N. 1968, *Tables of the Incomplete Beta Function* (Cambridge, UK: Cambridge University Press).

6.5 Bessel Functions of Integer Order

This section presents practical algorithms for computing various kinds of Bessel functions of integer order. In §6.6 we deal with fractional order. Actually, the more complicated routines for fractional order work fine for integer order too. For integer order, however, the routines in this section are simpler and faster.

For any real ν, the Bessel function $J_\nu(x)$ can be defined by the series representation

$$J_\nu(x) = \left(\frac{1}{2}x\right)^\nu \sum_{k=0}^{\infty} \frac{(-\frac{1}{4}x^2)^k}{k!\Gamma(\nu+k+1)} \tag{6.5.1}$$

The series converges for all x, but it is not computationally very useful for $x \gg 1$.

For ν *not* an integer, the Bessel function $Y_\nu(x)$ is given by

$$Y_\nu(x) = \frac{J_\nu(x)\cos(\nu\pi) - J_{-\nu}(x)}{\sin(\nu\pi)} \tag{6.5.2}$$

The right-hand side goes to the correct limiting value $Y_n(x)$ as ν goes to some integer n, but this is also not computationally useful.

For arguments $x < \nu$, both Bessel functions look qualitatively like simple power laws, with the asymptotic forms for $0 < x \ll \nu$

$$\begin{aligned} J_\nu(x) &\sim \frac{1}{\Gamma(\nu+1)}\left(\frac{1}{2}x\right)^\nu \qquad \nu \geq 0 \\ Y_0(x) &\sim \frac{2}{\pi}\ln(x) \\ Y_\nu(x) &\sim -\frac{\Gamma(\nu)}{\pi}\left(\frac{1}{2}x\right)^{-\nu} \qquad \nu > 0 \end{aligned} \tag{6.5.3}$$

For $x > \nu$, both Bessel functions look qualitatively like sine or cosine waves whose amplitude decays as $x^{-1/2}$. The asymptotic forms for $x \gg \nu$ are

$$\begin{aligned} J_\nu(x) &\sim \sqrt{\frac{2}{\pi x}}\cos\left(x - \frac{1}{2}\nu\pi - \frac{1}{4}\pi\right) \\ Y_\nu(x) &\sim \sqrt{\frac{2}{\pi x}}\sin\left(x - \frac{1}{2}\nu\pi - \frac{1}{4}\pi\right) \end{aligned} \tag{6.5.4}$$

In the transition region where $x \sim \nu$, the typical amplitudes of the Bessel functions are on the order

$$\begin{aligned} J_\nu(\nu) &\sim \frac{2^{1/3}}{3^{2/3}\Gamma(\frac{2}{3})}\frac{1}{\nu^{1/3}} \sim \frac{0.4473}{\nu^{1/3}} \\ Y_\nu(\nu) &\sim -\frac{2^{1/3}}{3^{1/6}\Gamma(\frac{2}{3})}\frac{1}{\nu^{1/3}} \sim -\frac{0.7748}{\nu^{1/3}} \end{aligned} \tag{6.5.5}$$

which holds asymptotically for large ν. Figure 6.5.1 plots the first few Bessel functions of each kind.

The Bessel functions satisfy the recurrence relations

$$J_{n+1}(x) = \frac{2n}{x}J_n(x) - J_{n-1}(x) \tag{6.5.6}$$

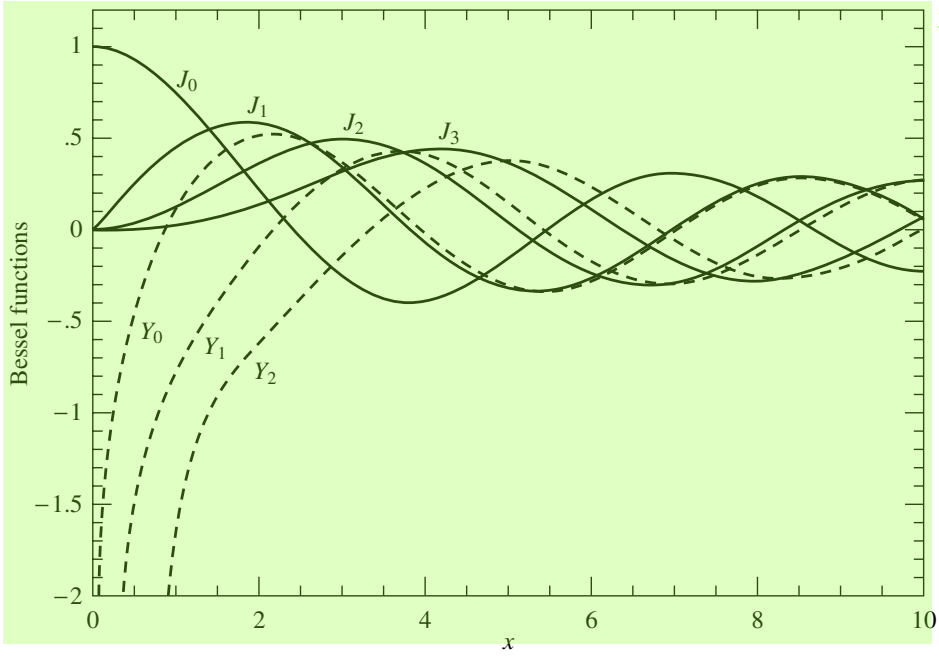

Figure 6.5.1. Bessel functions $J_0(x)$ through $J_3(x)$ and $Y_0(x)$ through $Y_2(x)$.

and

$$Y_{n+1}(x) = \frac{2n}{x} Y_n(x) - Y_{n-1}(x) \tag{6.5.7}$$

As already mentioned in §5.4, only the second of these, (6.5.7), is stable in the direction of increasing n for $x < n$. The reason that (6.5.6) is unstable in the direction of increasing n is simply that it is *the same recurrence* as (6.5.7): A small amount of "polluting" Y_n introduced by roundoff error will quickly come to swamp the desired J_n, according to equation (6.5.3).

A practical strategy for computing the Bessel functions of integer order divides into two tasks: first, how to compute J_0, J_1, Y_0, and Y_1; and second, how to use the recurrence relations stably to find other J's and Y's. We treat the first task first.

For x between zero and some arbitrary value (we will use the value 8), approximate $J_0(x)$ and $J_1(x)$ by rational functions in x. Likewise approximate by rational functions the "regular part" of $Y_0(x)$ and $Y_1(x)$, defined as

$$Y_0(x) - \frac{2}{\pi} J_0(x) \ln(x) \qquad \text{and} \qquad Y_1(x) - \frac{2}{\pi}\left[J_1(x) \ln(x) - \frac{1}{x} \right] \tag{6.5.8}$$

For $8 < x < \infty$, use the approximating forms ($n = 0, 1$)

$$J_n(x) = \sqrt{\frac{2}{\pi x}}\left[P_n\left(\frac{8}{x}\right)\cos(X_n) - Q_n\left(\frac{8}{x}\right)\sin(X_n) \right] \tag{6.5.9}$$

$$Y_n(x) = \sqrt{\frac{2}{\pi x}}\left[P_n\left(\frac{8}{x}\right)\sin(X_n) + Q_n\left(\frac{8}{x}\right)\cos(X_n) \right] \tag{6.5.10}$$

where

$$X_n \equiv x - \frac{2n+1}{4}\pi \tag{6.5.11}$$

and where P_0, P_1, Q_0, and Q_1 are each polynomials in their arguments, for $0 < 8/x < 1$. The P's are even polynomials, the Q's odd.

In the routines below, the various coefficients were calculated in multiple precision so as to achieve full double precision in the relative error. (In the neighborhood of the zeros of the functions, it is the absolute error that is double precision.) However, because of roundoff, evaluating the approximations can lead to a loss of up to two significant digits.

One additional twist: The rational approximation for $0 < x < 8$ is actually computed in the form [1]

$$J_0(x) = (x^2 - x_0^2)(x^2 - x_1^2)\frac{r(x^2)}{s(x^2)} \tag{6.5.12}$$

and similarly for J_1, Y_0, and Y_1. Here x_0 and x_1 are the two zeros of J_0 in the interval, and r and s are polynomials. The polynomial $r(x^2)$ has alternating signs. Writing it in terms of $64 - x^2$ makes all the signs the same and reduces roundoff error. For the approximations (6.5.9) and (6.5.10), our coefficients are similar but not identical to those given by Hart [2].

The functions J_0, J_1, Y_0, and Y_1 share a lot of code, so we package them as a single object `Bessjy`. The routines for higher J_n and Y_n are also member functions, with implementations discussed below. All the numerical coefficients are declared in `Bessjy` but defined (as a long list of constants) separately; the listing is in a Webnote [3].

bessel.h

```
struct Bessjy {
    static const Doub xj00,xj10,xj01,xj11,twoopi,pio4;
    static const Doub j0r[7],j0s[7],j0pn[5],j0pd[5],j0qn[5],j0qd[5];
    static const Doub j1r[7],j1s[7],j1pn[5],j1pd[5],j1qn[5],j1qd[5];
    static const Doub y0r[9],y0s[9],y0pn[5],y0pd[5],y0qn[5],y0qd[5];
    static const Doub y1r[8],y1s[8],y1pn[5],y1pd[5],y1qn[5],y1qd[5];
    Doub nump,denp,numq,denq,y,z,ax,xx;

    Doub j0(const Doub x) {
    Returns the Bessel function J0(x) for any real x.
        if ((ax=abs(x)) < 8.0) {          Direct rational function fit.
            rat(x,j0r,j0s,6);
            return nump*(y-xj00)*(y-xj10)/denp;
        } else {                          Fitting function (6.5.9).
            asp(j0pn,j0pd,j0qn,j0qd,1.);
            return sqrt(twoopi/ax)*(cos(xx)*nump/denp-z*sin(xx)*numq/denq);
        }
    }

    Doub j1(const Doub x) {
    Returns the Bessel function J1(x) for any real x.
        if ((ax=abs(x)) < 8.0) {          Direct rational approximation.
            rat(x,j1r,j1s,6);
            return x*nump*(y-xj01)*(y-xj11)/denp;
        } else {                          Fitting function (6.5.9).
            asp(j1pn,j1pd,j1qn,j1qd,3.);
            Doub ans=sqrt(twoopi/ax)*(cos(xx)*nump/denp-z*sin(xx)*numq/denq);
            return x > 0.0 ? ans : -ans;
        }
    }
```

```
Doub y0(const Doub x) {
Returns the Bessel function Y_0(x) for positive x.
    if (x < 8.0) {                      Rational function approximation of (6.5.8).
        Doub j0x = j0(x);
        rat(x,y0r,y0s,8);
        return nump/denp+twoopi*j0x*log(x);
    } else {                            Fitting function (6.5.10).
        ax=x;
        asp(y0pn,y0pd,y0qn,y0qd,1.);
        return sqrt(twoopi/x)*(sin(xx)*nump/denp+z*cos(xx)*numq/denq);
    }
}

Doub y1(const Doub x) {
Returns the Bessel function Y_1(x) for positive x.
    if (x < 8.0) {                      Rational function approximation of (6.5.8).
        Doub j1x = j1(x);
        rat(x,y1r,y1s,7);
        return x*nump/denp+twoopi*(j1x*log(x)-1.0/x);
    } else {                            Fitting function (6.5.10).
        ax=x;
        asp(y1pn,y1pd,y1qn,y1qd,3.);
        return sqrt(twoopi/x)*(sin(xx)*nump/denp+z*cos(xx)*numq/denq);
    }
}

Doub jn(const Int n, const Doub x);
Returns the Bessel function J_n(x) for any real x and integer n ≥ 0.

Doub yn(const Int n, const Doub x);
Returns the Bessel function Y_n(x) for any positive x and integer n ≥ 0.

void rat(const Doub x, const Doub *r, const Doub *s, const Int n) {
Common code: Evaluates rational approximation.
    y = x*x;
    z=64.0-y;
    nump=r[n];
    denp=s[n];
    for (Int i=n-1;i>=0;i--) {
        nump=nump*z+r[i];
        denp=denp*y+s[i];
    }
}

void asp(const Doub *pn, const Doub *pd, const Doub *qn, const Doub *qd,
Common code: Evaluates asymptotic approximation.
    const Doub fac) {
    z=8.0/ax;
    y=z*z;
    xx=ax-fac*pio4;
    nump=pn[4];
    denp=pd[4];
    numq=qn[4];
    denq=qd[4];
    for (Int i=3;i>=0;i--) {
        nump=nump*y+pn[i];
        denp=denp*y+pd[i];
        numq=numq*y+qn[i];
        denq=denq*y+qd[i];
    }
}
};
```

We now turn to the second task, namely, how to use the recurrence formulas (6.5.6) and (6.5.7) to get the Bessel functions $J_n(x)$ and $Y_n(x)$ for $n \geq 2$. The latter of these is straightforward, since its upward recurrence is always stable:

bessel.h

```
Doub Bessjy::yn(const Int n, const Doub x)
Returns the Bessel function Yn(x) for any positive x and integer n ≥ 0.
{
    Int j;
    Doub by,bym,byp,tox;
    if (n==0) return y0(x);
    if (n==1) return y1(x);
    tox=2.0/x;
    by=y1(x);                          Starting values for the recurrence.
    bym=y0(x);
    for (j=1;j<n;j++) {                Recurrence (6.5.7).
        byp=j*tox*by-bym;
        bym=by;
        by=byp;
    }
    return by;
}
```

The cost of this algorithm is the calls to `y1` and `y0` (which generate a call to each of `j1` and `j0`), plus $O(n)$ operations in the recurrence.

For $J_n(x)$ things are a bit more complicated. We can start the recurrence upward on n from J_0 and J_1 but it will remain stable only while n does not exceed x. This is, however, just fine for calls with large x and small n, a case that occurs frequently in practice.

The harder case to provide for is that with $x < n$. The best thing to do here is to use Miller's algorithm (see discussion preceding equation 5.4.16), applying the recurrence *downward* from some arbitrary starting value and making use of the upward-unstable nature of the recurrence to put us *onto* the correct solution. When we finally arrive at J_0 or J_1 we are able to normalize the solution with the sum (5.4.16) accumulated along the way.

The only subtlety is in deciding at how large an n we need start the downward recurrence so as to obtain a desired accuracy by the time we reach the n that we really want. If you play with the asymptotic forms (6.5.3) and (6.5.5), you should be able to convince yourself that the answer is to start larger than the desired n by an additive amount of order $[\text{constant} \times n]^{1/2}$, where the square root of the constant is, very roughly, the number of significant figures of accuracy.

The above considerations lead to the following function.

bessel.h

```
Doub Bessjy::jn(const Int n, const Doub x)
Returns the Bessel function Jn(x) for any real x and integer n ≥ 0.
{
    const Doub ACC=160.0;                  ACC determines accuracy.
    const Int IEXP=numeric_limits<Doub>::max_exponent/2;
    Bool jsum;
    Int j,k,m;
    Doub ax,bj,bjm,bjp,dum,sum,tox,ans;
    if (n==0) return j0(x);
    if (n==1) return j1(x);
    ax=abs(x);
    if (ax*ax <= 8.0*numeric_limits<Doub>::min()) return 0.0;
    else if (ax > Doub(n)) {               Upwards recurrence from J0 and J1.
        tox=2.0/ax;
```

```
        bjm=j0(ax);
        bj=j1(ax);
        for (j=1;j<n;j++) {
            bjp=j*tox*bj-bjm;
            bjm=bj;
            bj=bjp;
        }
        ans=bj;
    } else {                                Downward recurrence from an even m here
        tox=2.0/ax;                             computed.
        m=2*((n+Int(sqrt(ACC*n)))/2);
        jsum=false;                         jsum will alternate between false and true;
        bjp=ans=sum=0.0;                        when it is true, we accumulate in sum
        bj=1.0;                                 the even terms in (5.4.16).
        for (j=m;j>0;j--) {                 The downward recurrence.
            bjm=j*tox*bj-bjp;
            bjp=bj;
            bj=bjm;
            dum=frexp(bj,&k);
            if (k > IEXP) {                 Renormalize to prevent overflows.
                bj=ldexp(bj,-IEXP);
                bjp=ldexp(bjp,-IEXP);
                ans=ldexp(ans,-IEXP);
                sum=ldexp(sum,-IEXP);
            }
            if (jsum) sum += bj;            Accumulate the sum.
            jsum=!jsum;                     Change false to true or vice versa.
            if (j == n) ans=bjp;            Save the unnormalized answer.
        }
        sum=2.0*sum-bj;                     Compute (5.4.16)
        ans /= sum;                         and use it to normalize the answer.
    }
    return x < 0.0 && (n & 1) ? -ans : ans;
}
```

The function `ldexp`, used above, is a standard C and C++ library function for scaling the binary exponent of a number.

6.5.1 Modified Bessel Functions of Integer Order

The modified Bessel functions $I_n(x)$ and $K_n(x)$ are equivalent to the usual Bessel functions J_n and Y_n evaluated for purely imaginary arguments. In detail, the relationship is

$$\begin{aligned} I_n(x) &= (-i)^n J_n(ix) \\ K_n(x) &= \frac{\pi}{2} i^{n+1} [J_n(ix) + iY_n(ix)] \end{aligned} \tag{6.5.13}$$

The particular choice of prefactor and of the linear combination of J_n and Y_n to form K_n are simply choices that make the functions real-valued for real arguments x.

For small arguments $x \ll n$, both $I_n(x)$ and $K_n(x)$ become, asymptotically, simple powers of their arguments

$$\begin{aligned} I_n(x) &\approx \frac{1}{n!}\left(\frac{x}{2}\right)^n \qquad n \geq 0 \\ K_0(x) &\approx -\ln(x) \\ K_n(x) &\approx \frac{(n-1)!}{2}\left(\frac{x}{2}\right)^{-n} \qquad n > 0 \end{aligned} \tag{6.5.14}$$

These expressions are virtually identical to those for $J_n(x)$ and $Y_n(x)$ in this region, except for the factor of $-2/\pi$ difference between $Y_n(x)$ and $K_n(x)$. In the region $x \gg n$, however, the modified functions have quite different behavior than the Bessel functions,

$$I_n(x) \approx \frac{1}{\sqrt{2\pi x}} \exp(x)$$
$$K_n(x) \approx \frac{\pi}{\sqrt{2\pi x}} \exp(-x) \qquad (6.5.15)$$

The modified functions evidently have exponential rather than sinusoidal behavior for large arguments (see Figure 6.5.2). Rational approximations analogous to those for the J and Y Bessel functions are efficient for computing I_0, I_1, K_0, and K_1. The corresponding routines are packaged as an object `Bessik`. The routines are similar to those in [1], although different in detail. (All the constants are again listed in a Webnote [3].)

bessel.h

```
struct Bessik {
    static const Doub i0p[14],i0q[5],i0pp[5],i0qq[6];
    static const Doub i1p[14],i1q[5],i1pp[5],i1qq[6];
    static const Doub k0pi[5],k0qi[3],k0p[5],k0q[3],k0pp[8],k0qq[8];
    static const Doub k1pi[5],k1qi[3],k1p[5],k1q[3],k1pp[8],k1qq[8];
    Doub y,z,ax,term;

    Doub i0(const Doub x) {
    Returns the modified Bessel function I_0(x) for any real x.
        if ((ax=abs(x)) < 15.0) {          Rational approximation.
            y = x*x;
            return poly(i0p,13,y)/poly(i0q,4,225.-y);
        } else {                           Rational approximation with e^x/sqrt(x) factored out.
            z=1.0-15.0/ax;
            return exp(ax)*poly(i0pp,4,z)/(poly(i0qq,5,z)*sqrt(ax));
        }
    }

    Doub i1(const Doub x) {
    Returns the modified Bessel function I_1(x) for any real x.
        if ((ax=abs(x)) < 15.0) {          Rational approximation.
            y=x*x;
            return x*poly(i1p,13,y)/poly(i1q,4,225.-y);
        } else {                           Rational approximation with e^x/sqrt(x) factored out.
            z=1.0-15.0/ax;
            Doub ans=exp(ax)*poly(i1pp,4,z)/(poly(i1qq,5,z)*sqrt(ax));
            return x > 0.0 ? ans : -ans;
        }
    }

    Doub k0(const Doub x) {
    Returns the modified Bessel function K_0(x) for positive real x.
        if (x <= 1.0) {                    Use two rational approximations.
            z=x*x;
            term = poly(k0pi,4,z)*log(x)/poly(k0qi,2,1.-z);
            return poly(k0p,4,z)/poly(k0q,2,1.-z)-term;
        } else {                           Rational approximation with e^{-x}/sqrt(x) factored
            z=1.0/x;                           out.
            return exp(-x)*poly(k0pp,7,z)/(poly(k0qq,7,z)*sqrt(x));
        }
    }
```

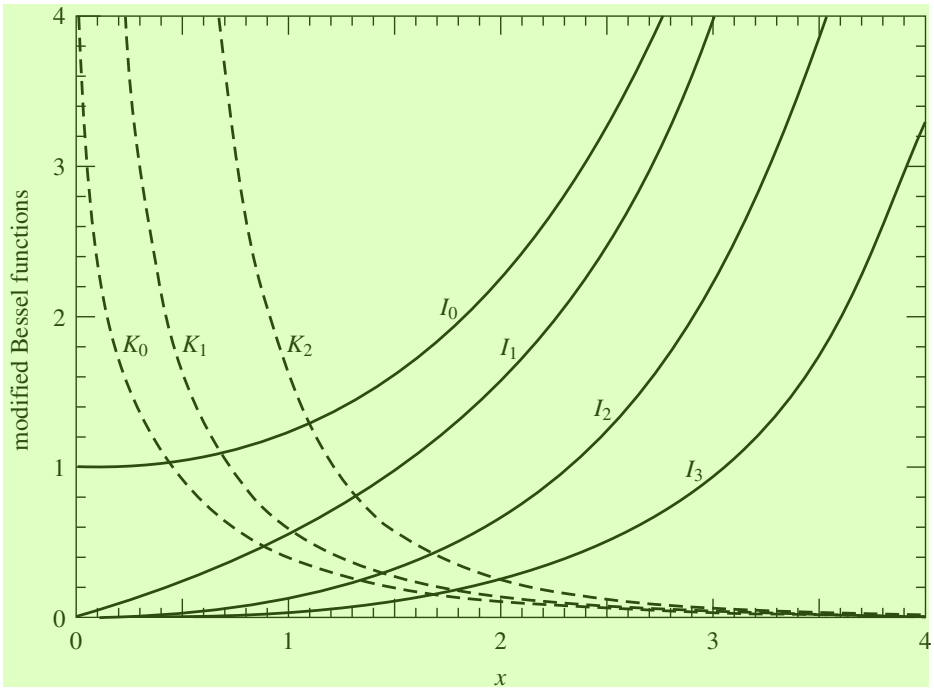

Figure 6.5.2. Modified Bessel functions $I_0(x)$ through $I_3(x)$, and $K_0(x)$ through $K_2(x)$.

```
Doub k1(const Doub x) {
Returns the modified Bessel function K1(x) for positive real x.
    if (x <= 1.0) {                    Use two rational approximations.
        z=x*x;
        term = poly(k1pi,4,z)*log(x)/poly(k1qi,2,1.-z);
        return x*(poly(k1p,4,z)/poly(k1q,2,1.-z)+term)+1./x;
    } else {                           Rational approximation with e^{-x}/sqrt(x) factored
        z=1.0/x;                           out.
        return exp(-x)*poly(k1pp,7,z)/(poly(k1qq,7,z)*sqrt(x));
    }
}

Doub in(const Int n, const Doub x);
Returns the modified Bessel function In(x) for any real x and n ≥ 0.

Doub kn(const Int n, const Doub x);
Returns the modified Bessel function Kn(x) for positive x and n ≥ 0.

inline Doub poly(const Doub *cof, const Int n, const Doub x) {
Common code: Evaluate a polynomial.
    Doub ans = cof[n];
    for (Int i=n-1;i>=0;i--) ans = ans*x+cof[i];
    return ans;
}
};
```

The recurrence relation for $I_n(x)$ and $K_n(x)$ is the same as that for $J_n(x)$ and $Y_n(x)$ provided that ix is substituted for x. This has the effect of changing a sign in the relation,

$$I_{n+1}(x) = -\left(\frac{2n}{x}\right) I_n(x) + I_{n-1}(x)$$
$$K_{n+1}(x) = +\left(\frac{2n}{x}\right) K_n(x) + K_{n-1}(x) \qquad (6.5.16)$$

These relations are always *unstable* for upward recurrence. For K_n itself growing, this presents no problem. The implementation is

bessel.h

```
Doub Bessik::kn(const Int n, const Doub x)
Returns the modified Bessel function K_n(x) for positive x and n ≥ 0.
{
    Int j;
    Doub bk,bkm,bkp,tox;
    if (n==0) return k0(x);
    if (n==1) return k1(x);
    tox=2.0/x;
    bkm=k0(x);                          Upward recurrence for all x...
    bk=k1(x);
    for (j=1;j<n;j++) {                 ...and here it is.
        bkp=bkm+j*tox*bk;
        bkm=bk;
        bk=bkp;
    }
    return bk;
}
```

For I_n, the strategy of downward recursion is required once again, and the starting point for the recursion may be chosen in the same manner as for the routine `Bessjy::jn`. The only fundamental difference is that the normalization formula for $I_n(x)$ has an alternating minus sign in successive terms, which again arises from the substitution of ix for x in the formula used previously for J_n:

$$1 = I_0(x) - 2I_2(x) + 2I_4(x) - 2I_6(x) + \cdots \qquad (6.5.17)$$

In fact, we prefer simply to normalize with a call to `i0`.

bessel.h

```
Doub Bessik::in(const Int n, const Doub x)
Returns the modified Bessel function I_n(x) for any real x and n ≥ 0.
{
    const Doub ACC=200.0;                   ACC determines accuracy.
    const Int IEXP=numeric_limits<Doub>::max_exponent/2;
    Int j,k;
    Doub bi,bim,bip,dum,tox,ans;
    if (n==0) return i0(x);
    if (n==1) return i1(x);
    if (x*x <= 8.0*numeric_limits<Doub>::min()) return 0.0;
    else {
        tox=2.0/abs(x);
        bip=ans=0.0;
        bi=1.0;
        for (j=2*(n+Int(sqrt(ACC*n)));j>0;j--) {      Downward recurrence.
            bim=bip+j*tox*bi;
            bip=bi;
            bi=bim;
            dum=frexp(bi,&k);
            if (k > IEXP) {                 Renormalize to prevent overflows.
                ans=ldexp(ans,-IEXP);
                bi=ldexp(bi,-IEXP);
                bip=ldexp(bip,-IEXP);
```

```
            }
            if (j == n) ans=bip;
        }
        ans *= i0(x)/bi;                    Normalize with bessi0.
        return x < 0.0 && (n & 1) ? -ans : ans;
    }
}
```

The function `ldexp`, used above, is a standard C and C++ library function for scaling the binary exponent of a number.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at `http://www.nr.com/aands`, Chapter 9.

Carrier, G.F., Krook, M. and Pearson, C.E. 1966, *Functions of a Complex Variable* (New York: McGraw-Hill), pp. 220ff.

SPECFUN, 2007+, at `http://www.netlib.org/specfun`.[1]

Hart, J.F., et al. 1968, *Computer Approximations* (New York: Wiley), §6.8, p. 141.[2]

Numerical Recipes Software 2007, "Coefficients Used in the Bessjy and Bessik Objects," *Numerical Recipes Webnote No. 7*, at `http://www.nr.com/webnotes?7` [3]

6.6 Bessel Functions of Fractional Order, Airy Functions, Spherical Bessel Functions

Many algorithms have been proposed for computing Bessel functions of fractional order numerically. Most of them are, in fact, not very good in practice. The routines given here are rather complicated, but they can be recommended wholeheartedly.

6.6.1 Ordinary Bessel Functions

The basic idea is *Steed's method*, which was originally developed [1] for Coulomb wave functions. The method calculates J_ν, J'_ν, Y_ν, and Y'_ν simultaneously, and so involves four relations among these functions. Three of the relations come from two continued fractions, one of which is complex. The fourth is provided by the Wronskian relation

$$W \equiv J_\nu Y'_\nu - Y_\nu J'_\nu = \frac{2}{\pi x} \tag{6.6.1}$$

The first continued fraction, CF1, is defined by

$$f_\nu \equiv \frac{J'_\nu}{J_\nu} = \frac{\nu}{x} - \frac{J_{\nu+1}}{J_\nu} = \frac{\nu}{x} - \frac{1}{2(\nu+1)/x -} \; \frac{1}{2(\nu+2)/x -} \cdots \tag{6.6.2}$$

You can easily derive it from the three-term recurrence relation for Bessel functions: Start with equation (6.5.6) and use equation (5.4.18). Forward evaluation of the continued fraction by one of the methods of §5.2 is essentially equivalent to backward recurrence of the recurrence relation. The rate of convergence of CF1 is determined by the position of the *turning point* $x_{\rm tp} = \sqrt{\nu(\nu+1)} \approx \nu$ beyond which the Bessel functions become oscillatory. If $x \lesssim x_{\rm tp}$,

convergence is very rapid. If $x \gtrsim x_{\rm tp}$, then each iteration of the continued fraction effectively increases ν by one until $x \lesssim x_{\rm tp}$; thereafter rapid convergence sets in. Thus the number of iterations of CF1 is of order x for large x. In the routine `besseljy` we set the maximum allowed number of iterations to 10,000. For larger x, you can use the usual asymptotic expressions for Bessel functions.

One can show that the sign of J_ν is the same as the sign of the denominator of CF1 once it has converged.

The complex continued fraction CF2 is defined by

$$p + iq \equiv \frac{J_\nu' + iY_\nu'}{J_\nu + iY_\nu} = -\frac{1}{2x} + i + \frac{i}{x}\,\frac{(1/2)^2 - \nu^2}{2(x+i)+}\,\frac{(3/2)^2 - \nu^2}{2(x+2i)+}\cdots \tag{6.6.3}$$

(We sketch the derivation of CF2 in the analogous case of modified Bessel functions in the next subsection.) This continued fraction converges rapidly for $x \gtrsim x_{\rm tp}$, while convergence fails as $x \to 0$. We have to adopt a special method for small x, which we describe below. For x not too small, we can ensure that $x \gtrsim x_{\rm tp}$ by a stable recurrence of J_ν and J_ν' downward to a value $\nu = \mu \lesssim x$, thus yielding the ratio f_μ of this lower value of ν. This is the stable direction for the recurrence relation. The initial values for the recurrence are

$$J_\nu = \text{arbitrary}, \qquad J_\nu' = f_\nu J_\nu, \tag{6.6.4}$$

with the sign of the arbitrary initial value of J_ν chosen to be the sign of the denominator of CF1. Choosing the initial value of J_ν very small minimizes the possibility of overflow during the recurrence. The recurrence relations are

$$\begin{aligned} J_{\nu-1} &= \frac{\nu}{x} J_\nu + J_\nu' \\ J_{\nu-1}' &= \frac{\nu - 1}{x} J_{\nu-1} - J_\nu \end{aligned} \tag{6.6.5}$$

Once CF2 has been evaluated at $\nu = \mu$, then with the Wronskian (6.6.1) we have enough relations to solve for all four quantities. The formulas are simplified by introducing the quantity

$$\gamma \equiv \frac{p - f_\mu}{q} \tag{6.6.6}$$

Then

$$J_\mu = \pm\left(\frac{W}{q + \gamma(p - f_\mu)}\right)^{1/2} \tag{6.6.7}$$

$$J_\mu' = f_\mu J_\mu \tag{6.6.8}$$

$$Y_\mu = \gamma J_\mu \tag{6.6.9}$$

$$Y_\mu' = Y_\mu\left(p + \frac{q}{\gamma}\right) \tag{6.6.10}$$

The sign of J_μ in (6.6.7) is chosen to be the same as the sign of the initial J_ν in (6.6.4).

Once all four functions have been determined at the value $\nu = \mu$, we can find them at the original value of ν. For J_ν and J_ν' simply scale the values in (6.6.4) by the ratio of (6.6.7) to the value found after applying the recurrence (6.6.5). The quantities Y_ν and Y_ν' can be found by starting with the values in (6.6.9) and (6.6.10) and using the stable upward recurrence

$$Y_{\nu+1} = \frac{2\nu}{x} Y_\nu - Y_{\nu-1} \tag{6.6.11}$$

together with the relation

$$Y_\nu' = \frac{\nu}{x} Y_\nu - Y_{\nu+1} \tag{6.6.12}$$

Now turn to the case of small x, when CF2 is not suitable. Temme [2] has given a good method of evaluating Y_ν and $Y_{\nu+1}$, and hence Y_ν' from (6.6.12), by series expansions that

accurately handle the singularity as $x \to 0$. The expansions work only for $|\nu| \le 1/2$, and so now the recurrence (6.6.5) is used to evaluate f_ν at a value $\nu = \mu$ in this interval. Then one calculates J_μ from

$$J_\mu = \frac{W}{Y'_\mu - Y_\mu f_\mu} \tag{6.6.13}$$

and J'_μ from (6.6.8). The values at the original value of ν are determined by scaling as before, and the Y's are recurred up as before.

Temme's series are

$$Y_\nu = -\sum_{k=0}^{\infty} c_k g_k \qquad Y_{\nu+1} = -\frac{2}{x}\sum_{k=0}^{\infty} c_k h_k \tag{6.6.14}$$

Here

$$c_k = \frac{(-x^2/4)^k}{k!} \tag{6.6.15}$$

while the coefficients g_k and h_k are defined in terms of quantities p_k, q_k, and f_k that can be found by recursion:

$$\begin{aligned} g_k &= f_k + \frac{2}{\nu}\sin^2\left(\frac{\nu\pi}{2}\right) q_k \\ h_k &= -k g_k + p_k \\ p_k &= \frac{p_{k-1}}{k-\nu} \\ q_k &= \frac{q_{k-1}}{k+\nu} \\ f_k &= \frac{k f_{k-1} + p_{k-1} + q_{k-1}}{k^2 - \nu^2} \end{aligned} \tag{6.6.16}$$

The initial values for the recurrences are

$$\begin{aligned} p_0 &= \frac{1}{\pi}\left(\frac{x}{2}\right)^{-\nu}\Gamma(1+\nu) \\ q_0 &= \frac{1}{\pi}\left(\frac{x}{2}\right)^{\nu}\Gamma(1-\nu) \\ f_0 &= \frac{2}{\pi}\frac{\nu\pi}{\sin\nu\pi}\left[\cosh\sigma\,\Gamma_1(\nu) + \frac{\sinh\sigma}{\sigma}\ln\left(\frac{2}{x}\right)\Gamma_2(\nu)\right] \end{aligned} \tag{6.6.17}$$

with

$$\begin{aligned} \sigma &= \nu\ln\left(\frac{2}{x}\right) \\ \Gamma_1(\nu) &= \frac{1}{2\nu}\left[\frac{1}{\Gamma(1-\nu)} - \frac{1}{\Gamma(1+\nu)}\right] \\ \Gamma_2(\nu) &= \frac{1}{2}\left[\frac{1}{\Gamma(1-\nu)} + \frac{1}{\Gamma(1+\nu)}\right] \end{aligned} \tag{6.6.18}$$

The whole point of writing the formulas in this way is that the potential problems as $\nu \to 0$ can be controlled by evaluating $\nu\pi/\sin\nu\pi$, $\sinh\sigma/\sigma$, and Γ_1 carefully. In particular, Temme gives Chebyshev expansions for $\Gamma_1(\nu)$ and $\Gamma_2(\nu)$. We have rearranged his expansion for Γ_1 to be explicitly an even series in ν for more efficient evaluation, as explained in §5.8.

Because J_ν, Y_ν, J'_ν, and Y'_ν are all calculated simultaneously, a single `void` function sets them all. You then grab those that you need directly from the object. Alternatively, the functions `jnu` and `ynu` can be used. (We've omitted similar helper functions for the derivatives, but you can easily add them.) The object `Bessel` contains various other methods that will be discussed below.

The routines assume $\nu \geq 0$. For negative ν you can use the reflection formulas

$$J_{-\nu} = \cos \nu\pi \; J_\nu - \sin \nu\pi \; Y_\nu$$
$$Y_{-\nu} = \sin \nu\pi \; J_\nu + \cos \nu\pi \; Y_\nu \qquad (6.6.19)$$

The routine also assumes $x > 0$. For $x < 0$, the functions are in general complex but expressible in terms of functions with $x > 0$. For $x = 0$, Y_ν is singular. The complex arithmetic is carried out explicitly with real variables.

besselfrac.h

```
struct Bessel {
```
Object for Bessel functions of arbitrary order ν, and related functions.

```
    static const Int NUSE1=7, NUSE2=8;
    static const Doub c1[NUSE1],c2[NUSE2];
    Doub xo,nuo;                                  Saved x and ν from last call.
    Doub jo,yo,jpo,ypo;                           Set by besseljy.
    Doub io,ko,ipo,kpo;                           Set by besselik.
    Doub aio,bio,aipo,bipo;                       Set by airy.
    Doub sphjo,sphyo,sphjpo,sphypo;               Set by sphbes.
    Int sphno;

    Bessel() : xo(9.99e99), nuo(9.99e99), sphno(-9999) {}
```
Default constructor. No arguments.

```
    void besseljy(const Doub nu, const Doub x);
```
Calculate Bessel functions $J_\nu(x)$ and $Y_\nu(x)$ and their derivatives.

```
    void besselik(const Doub nu, const Doub x);
```
Calculate Bessel functions $I_\nu(x)$ and $K_\nu(x)$ and their derivatives.

```
    Doub jnu(const Doub nu, const Doub x) {
```
Simple interface returning $J_\nu(x)$.

```
        if (nu != nuo || x != xo) besseljy(nu,x);
        return jo;
    }
    Doub ynu(const Doub nu, const Doub x) {
```
Simple interface returning $Y_\nu(x)$.

```
        if (nu != nuo || x != xo) besseljy(nu,x);
        return yo;
    }
    Doub inu(const Doub nu, const Doub x) {
```
Simple interface returning $I_\nu(x)$.

```
        if (nu != nuo || x != xo) besselik(nu,x);
        return io;
    }
    Doub knu(const Doub nu, const Doub x) {
```
Simple interface returning $K_\nu(x)$.

```
        if (nu != nuo || x != xo) besselik(nu,x);
        return ko;
    }

    void airy(const Doub x);
```
Calculate Airy functions $\mathrm{Ai}(x)$ and $\mathrm{Bi}(x)$ and their derivatives.

```
    Doub airy_ai(const Doub x);
```
Simple interface returning $\mathrm{Ai}(x)$.

```
    Doub airy_bi(const Doub x);
```
Simple interface returning $\mathrm{Bi}(x)$.

```
    void sphbes(const Int n, const Doub x);
```
Calculate spherical Bessel functions $j_n(x)$ and $y_n(x)$ and their derivatives.

```
    Doub sphbesj(const Int n, const Doub x);
```
Simple interface returning $j_n(x)$.

```
    Doub sphbesy(const Int n, const Doub x);
```
Simple interface returning $y_n(x)$.

```
    inline Doub chebev(const Doub *c, const Int m, const Doub x) {
    Utility used by besseljy and besselik, evaluates Chebyshev series.
        Doub d=0.0,dd=0.0,sv;
        Int j;
        for (j=m-1;j>0;j--) {
            sv=d;
            d=2.*x*d-dd+c[j];
            dd=sv;
        }
        return x*d-dd+0.5*c[0];
    }
};

const Doub Bessel::c1[7] = {-1.142022680371168e0,6.5165112670737e-3,
    3.087090173086e-4,-3.4706269649e-6,6.9437664e-9,3.67795e-11,
    -1.356e-13};
const Doub Bessel::c2[8] = {1.843740587300905e0,-7.68528408447867e-2,
    1.2719271366546e-3,-4.9717367042e-6,-3.31261198e-8,2.423096e-10,
    -1.702e-13,-1.49e-15};
```

The code listing for `Bessel::besseljy` is in a Webnote [4].

6.6.2 Modified Bessel Functions

Steed's method does not work for modified Bessel functions because in this case CF2 is purely imaginary and we have only three relations among the four functions. Temme [3] has given a normalization condition that provides the fourth relation.

The Wronskian relation is

$$W \equiv I_\nu K'_\nu - K_\nu I'_\nu = -\frac{1}{x} \tag{6.6.20}$$

The continued fraction CF1 becomes

$$f_\nu \equiv \frac{I'_\nu}{I_\nu} = \frac{\nu}{x} + \frac{1}{2(\nu+1)/x +} \; \frac{1}{2(\nu+2)/x +} \cdots \tag{6.6.21}$$

To get CF2 and the normalization condition in a convenient form, consider the sequence of confluent hypergeometric functions

$$z_n(x) = U(\nu + 1/2 + n, 2\nu + 1, 2x) \tag{6.6.22}$$

for fixed ν. Then

$$K_\nu(x) = \pi^{1/2}(2x)^\nu e^{-x} z_0(x) \tag{6.6.23}$$

$$\frac{K_{\nu+1}(x)}{K_\nu(x)} = \frac{1}{x}\left[\nu + \frac{1}{2} + x + \left(\nu^2 - \frac{1}{4}\right)\frac{z_1}{z_0}\right] \tag{6.6.24}$$

Equation (6.6.23) is the standard expression for K_ν in terms of a confluent hypergeometric function, while equation (6.6.24) follows from relations between contiguous confluent hypergeometric functions (equations 13.4.16 and 13.4.18 in Ref. [5]). Now the functions z_n satisfy the three-term recurrence relation (equation 13.4.15 in Ref. [5])

$$z_{n-1}(x) = b_n z_n(x) + a_{n+1} z_{n+1} \tag{6.6.25}$$

with

$$\begin{aligned} b_n &= 2(n+x) \\ a_{n+1} &= -[(n+1/2)^2 - \nu^2] \end{aligned} \tag{6.6.26}$$

Following the steps leading to equation (5.4.18), we get the continued fraction CF2

$$\frac{z_1}{z_0} = \frac{1}{b_1 +} \; \frac{a_2}{b_2 +} \cdots \tag{6.6.27}$$

from which (6.6.24) gives $K_{\nu+1}/K_\nu$ and thus K'_ν/K_ν.

Temme's normalization condition is that

$$\sum_{n=0}^{\infty} C_n z_n = \left(\frac{1}{2x}\right)^{\nu+1/2} \tag{6.6.28}$$

where

$$C_n = \frac{(-1)^n}{n!}\frac{\Gamma(\nu+1/2+n)}{\Gamma(\nu+1/2-n)} \tag{6.6.29}$$

Note that the C_n's can be determined by recursion:

$$C_0 = 1, \qquad C_{n+1} = -\frac{a_{n+1}}{n+1}C_n \tag{6.6.30}$$

We use the condition (6.6.28) by finding

$$S = \sum_{n=1}^{\infty} C_n \frac{z_n}{z_0} \tag{6.6.31}$$

Then

$$z_0 = \left(\frac{1}{2x}\right)^{\nu+1/2} \frac{1}{1+S} \tag{6.6.32}$$

and (6.6.23) gives K_ν.

Thompson and Barnett [6] have given a clever method of doing the sum (6.6.31) simultaneously with the forward evaluation of the continued fraction CF2. Suppose the continued fraction is being evaluated as

$$\frac{z_1}{z_0} = \sum_{n=0}^{\infty} \Delta h_n \tag{6.6.33}$$

where the increments Δh_n are being found by, e.g., Steed's algorithm or the modified Lentz's algorithm of §5.2. Then the approximation to S keeping the first N terms can be found as

$$S_N = \sum_{n=1}^{N} Q_n \Delta h_n \tag{6.6.34}$$

Here

$$Q_n = \sum_{k=1}^{n} C_k q_k \tag{6.6.35}$$

and q_k is found by recursion from

$$q_{k+1} = (q_{k-1} - b_k q_k)/a_{k+1} \tag{6.6.36}$$

starting with $q_0 = 0, q_1 = 1$. For the case at hand, approximately three times as many terms are needed to get S to converge as are needed simply for CF2 to converge.

To find K_ν and $K_{\nu+1}$ for small x we use series analogous to (6.6.14):

$$K_\nu = \sum_{k=0}^{\infty} c_k f_k \qquad K_{\nu+1} = \frac{2}{x}\sum_{k=0}^{\infty} c_k h_k \tag{6.6.37}$$

Here

$$
\begin{aligned}
c_k &= \frac{(x^2/4)^k}{k!} \\
h_k &= -k f_k + p_k \\
p_k &= \frac{p_{k-1}}{k-\nu} \\
q_k &= \frac{q_{k-1}}{k+\nu} \\
f_k &= \frac{k f_{k-1} + p_{k-1} + q_{k-1}}{k^2 - \nu^2}
\end{aligned}
\tag{6.6.38}
$$

The initial values for the recurrences are

$$
\begin{aligned}
p_0 &= \frac{1}{2}\left(\frac{x}{2}\right)^{-\nu} \Gamma(1+\nu) \\
q_0 &= \frac{1}{2}\left(\frac{x}{2}\right)^{\nu} \Gamma(1-\nu) \\
f_0 &= \frac{\nu\pi}{\sin\nu\pi}\left[\cosh\sigma\,\Gamma_1(\nu) + \frac{\sinh\sigma}{\sigma}\ln\left(\frac{2}{x}\right)\Gamma_2(\nu)\right]
\end{aligned}
\tag{6.6.39}
$$

Both the series for small x, and CF2 and the normalization relation (6.6.28) require $|\nu| \le 1/2$. In both cases, therefore, we recurse I_ν down to a value $\nu = \mu$ in this interval, find K_μ there, and recurse K_ν back up to the original value of ν.

The routine assumes $\nu \ge 0$. For negative ν use the reflection formulas

$$
\begin{aligned}
I_{-\nu} &= I_\nu + \frac{2}{\pi}\sin(\nu\pi)\,K_\nu \\
K_{-\nu} &= K_\nu
\end{aligned}
\tag{6.6.40}
$$

Note that for large x, $I_\nu \sim e^x$ and $K_\nu \sim e^{-x}$, and so these functions will overflow or underflow. It is often desirable to be able to compute the scaled quantities $e^{-x} I_\nu$ and $e^x K_\nu$. Simply omitting the factor e^{-x} in equation (6.6.23) will ensure that all four quantities will have the appropriate scaling. If you also want to scale the four quantities for small x when the series in equation (6.6.37) are used, you must multiply each series by e^x.

As with `besseljy`, you can either call the `void` function `besselik`, and then retrieve the function and/or derivative values from the object, or else just call `inu` or `knu`.

The code listing for `Bessel::besselik` is in a Webnote [4].

6.6.3 Airy Functions

For positive x, the Airy functions are defined by

$$
\mathrm{Ai}(x) = \frac{1}{\pi}\sqrt{\frac{x}{3}}K_{1/3}(z)
\tag{6.6.41}
$$

$$
\mathrm{Bi}(x) = \sqrt{\frac{x}{3}}[I_{1/3}(z) + I_{-1/3}(z)]
\tag{6.6.42}
$$

where

$$
z = \frac{2}{3}x^{3/2}
\tag{6.6.43}
$$

By using the reflection formula (6.6.40), we can convert (6.6.42) into the computationally more useful form

$$
\mathrm{Bi}(x) = \sqrt{x}\left[\frac{2}{\sqrt{3}}I_{1/3}(z) + \frac{1}{\pi}K_{1/3}(z)\right]
\tag{6.6.44}
$$

so that Ai and Bi can be evaluated with a single call to `besselik`.

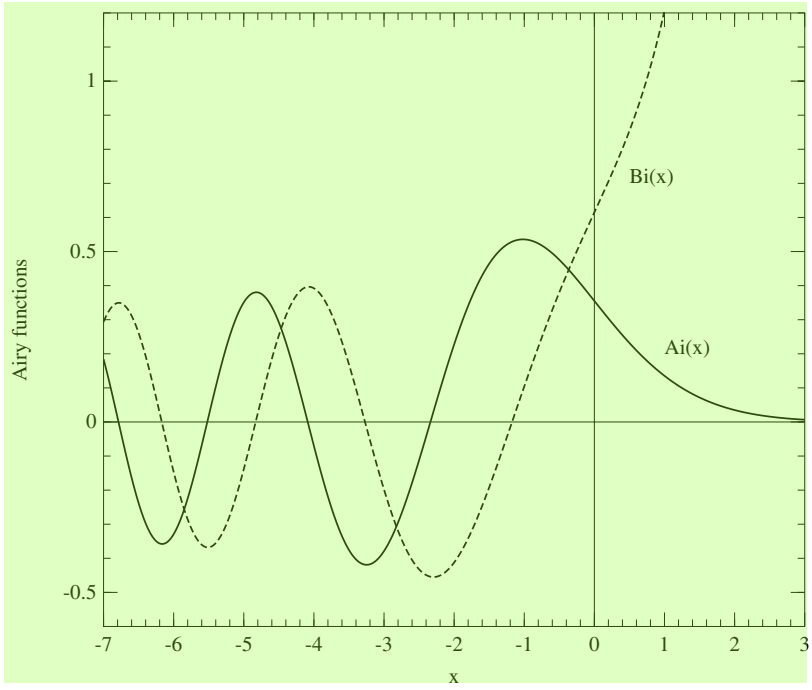

Figure 6.6.1. Airy functions $\mathrm{Ai}(x)$ and $\mathrm{Bi}(x)$.

The derivatives should not be evaluated by simply differentiating the above expressions because of possible subtraction errors near $x = 0$. Instead, use the equivalent expressions

$$\begin{aligned} \mathrm{Ai}'(x) &= -\frac{x}{\pi\sqrt{3}} K_{2/3}(z) \\ \mathrm{Bi}'(x) &= x\left[\frac{2}{\sqrt{3}} I_{2/3}(z) + \frac{1}{\pi} K_{2/3}(z)\right] \end{aligned} \tag{6.6.45}$$

The corresponding formulas for negative arguments are

$$\begin{aligned} \mathrm{Ai}(-x) &= \frac{\sqrt{x}}{2}\left[J_{1/3}(z) - \frac{1}{\sqrt{3}} Y_{1/3}(z)\right] \\ \mathrm{Bi}(-x) &= -\frac{\sqrt{x}}{2}\left[\frac{1}{\sqrt{3}} J_{1/3}(z) + Y_{1/3}(z)\right] \\ \mathrm{Ai}'(-x) &= \frac{x}{2}\left[J_{2/3}(z) + \frac{1}{\sqrt{3}} Y_{2/3}(z)\right] \\ \mathrm{Bi}'(-x) &= \frac{x}{2}\left[\frac{1}{\sqrt{3}} J_{2/3}(z) - Y_{2/3}(z)\right] \end{aligned} \tag{6.6.46}$$

besselfrac.h

```
void Bessel::airy(const Doub x) {
```

Sets aio, bio, aipo, and bipo, respectively, to the Airy functions $\mathrm{Ai}(x)$, $\mathrm{Bi}(x)$ and their derivatives $\mathrm{Ai}'(x)$, $\mathrm{Bi}'(x)$.

```
    static const Doub PI=3.141592653589793238,
        ONOVRT=0.577350269189626,THR=1./3.,TWOTHR=2.*THR;
    Doub absx,rootx,z;
    absx=abs(x);
    rootx=sqrt(absx);
    z=TWOTHR*absx*rootx;
```

```
    if (x > 0.0) {
        besselik(THR,z);
        aio = rootx*ONOVRT*ko/PI;
        bio = rootx*(ko/PI+2.0*ONOVRT*io);
        besselik(TWOTHR,z);
        aipo = -x*ONOVRT*ko/PI;
        bipo = x*(ko/PI+2.0*ONOVRT*io);
    } else if (x < 0.0) {
        besseljy(THR,z);
        aio = 0.5*rootx*(jo-ONOVRT*yo);
        bio = -0.5*rootx*(yo+ONOVRT*jo);
        besseljy(TWOTHR,z);
        aipo = 0.5*absx*(ONOVRT*yo+jo);
        bipo = 0.5*absx*(ONOVRT*jo-yo);
    } else {
        aio=0.355028053887817;
        bio=aio/ONOVRT;
        aipo = -0.258819403792807;
        bipo = -aipo/ONOVRT;
    }
}

Doub Bessel::airy_ai(const Doub x) {
    if (x != xo) airy(x);
    return aio;
}
Doub Bessel::airy_bi(const Doub x) {
    if (x != xo) airy(x);
    return bio;
}
```

Case $x = 0$.

Simple interface returning $\mathrm{Ai}(x)$.

Simple interface returning $\mathrm{Bi}(x)$.

6.6.4 Spherical Bessel Functions

For integer n, spherical Bessel functions are defined by

$$j_n(x) = \sqrt{\frac{\pi}{2x}} J_{n+\frac{1}{2}}(x)$$
$$y_n(x) = \sqrt{\frac{\pi}{2x}} Y_{n+\frac{1}{2}}(x) \tag{6.6.47}$$

They can be evaluated by a call to `besseljy`, and the derivatives can safely be found from the derivatives of equation (6.6.47).

Note that in the continued fraction CF2 in (6.6.3) just the first term survives for $\nu = 1/2$. Thus one can make a very simple algorithm for spherical Bessel functions along the lines of `besseljy` by always recursing j_n down to $n = 0$, setting p and q from the first term in CF2, and then recursing y_n up. No special series is required near $x = 0$. However, `besseljy` is already so efficient that we have not bothered to provide an independent routine for spherical Bessels.

```
void Bessel::sphbes(const Int n, const Doub x) {
    const Doub RTPIO2=1.253314137315500251;
    Doub factor,order;
    if (n < 0 || x <= 0.0) throw("bad arguments in sphbes");
    order=n+0.5;
    besseljy(order,x);
    factor=RTPIO2/sqrt(x);
    sphjo=factor*jo;
```

Sets `sphjo`, `sphyo`, `sphjpo`, and `sphypo`, respectively, to the spherical Bessel functions $j_n(x)$, $y_n(x)$, and their derivatives $j_n'(x)$, $y_n'(x)$ for integer n (which is saved as `sphno`).

besselfrac.h

```
    sphyo=factor*yo;
    sphjpo=factor*jpo-sphjo/(2.*x);
    sphypo=factor*ypo-sphyo/(2.*x);
    sphno = n;
}

Doub Bessel::sphbesj(const Int n, const Doub x) {
Simple interface returning j_n(x).
    if (n != sphno || x != xo) sphbes(n,x);
    return sphjo;
}
Doub Bessel::sphbesy(const Int n, const Doub x) {
Simple interface returning y_n(x).
    if (n != sphno || x != xo) sphbes(n,x);
    return sphyo;
}
```

CITED REFERENCES AND FURTHER READING:

Barnett, A.R., Feng, D.H., Steed, J.W., and Goldfarb, L.J.B. 1974, "Coulomb Wave Functions for All Real η and ρ," *Computer Physics Communications*, vol. 8, pp. 377–395.[1]

Temme, N.M. 1976, "On the Numerical Evaluation of the Ordinary Bessel Function of the Second Kind," *Journal of Computational Physics*, vol. 21, pp. 343–350[2]; 1975, *op. cit.*, vol. 19, pp. 324–337.[3]

Numerical Recipes Software 2007, "Bessel Function Implementations," *Numerical Recipes Webnote No. 8*, at `http://www.nr.com/webnotes?8` [4]

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at `http://www.nr.com/aands`, Chapter 10.[5]

Thompson, I.J., and Barnett, A.R. 1987, "Modified Bessel Functions $I_\nu(z)$ and $K_\nu(z)$ of Real Order and Complex Argument, to Selected Accuracy," *Computer Physics Communications*, vol. 47, pp. 245–257.[6]

Barnett, A.R. 1981, "An Algorithm for Regular and Irregular Coulomb and Bessel functions of Real Order to Machine Accuracy," *Computer Physics Communications*, vol. 21, pp. 297–314.

Thompson, I.J., and Barnett, A.R. 1986, "Coulomb and Bessel Functions of Complex Arguments and Order," *Journal of Computational Physics*, vol. 64, pp. 490–509.

6.7 Spherical Harmonics

Spherical harmonics occur in a large variety of physical problems, for example, whenever a wave equation, or Laplace's equation, is solved by separation of variables in spherical coordinates. The spherical harmonic $Y_{lm}(\theta,\phi), -l \le m \le l$, is a function of the two coordinates θ, ϕ on the surface of a sphere.

The spherical harmonics are orthogonal for different l and m, and they are normalized so that their integrated square over the sphere is unity:

$$\int_0^{2\pi} d\phi \int_{-1}^{1} d(\cos\theta) Y^*_{l'm'}(\theta,\phi) Y_{lm}(\theta,\phi) = \delta_{l'l}\delta_{m'm} \tag{6.7.1}$$

Here the asterisk denotes complex conjugation.

Mathematically, the spherical harmonics are related to *associated Legendre polynomials* by the equation

$$Y_{lm}(\theta,\phi) = \sqrt{\frac{2l+1}{4\pi}\frac{(l-m)!}{(l+m)!}}\,P_l^m(\cos\theta)e^{im\phi} \tag{6.7.2}$$

By using the relation

$$Y_{l,-m}(\theta,\phi) = (-1)^m Y_{lm}^*(\theta,\phi) \tag{6.7.3}$$

we can always relate a spherical harmonic to an associated Legendre polynomial with $m \geq 0$. With $x \equiv \cos\theta$ these are defined in terms of the ordinary Legendre polynomials (cf. §4.6 and §5.4) by

$$P_l^m(x) = (-1)^m(1-x^2)^{m/2}\frac{d^m}{dx^m}P_l(x) \tag{6.7.4}$$

Be careful: There are alternative normalizations for the associated Legendre polynomials and alternative sign conventions.

The first few associated Legendre polynomials, and their corresponding normalized spherical harmonics, are

$$\begin{array}{ll|ll}
P_0^0(x) = & 1 & Y_{00} = & \sqrt{\frac{1}{4\pi}} \\
\hline
P_1^1(x) = & -(1-x^2)^{1/2} & Y_{11} = & -\sqrt{\frac{3}{8\pi}}\sin\theta e^{i\phi} \\
P_1^0(x) = & x & Y_{10} = & \sqrt{\frac{3}{4\pi}}\cos\theta \\
\hline
P_2^2(x) = & 3(1-x^2) & Y_{22} = & \tfrac{1}{4}\sqrt{\frac{15}{2\pi}}\sin^2\theta e^{2i\phi} \\
P_2^1(x) = & -3(1-x^2)^{1/2}x & Y_{21} = & -\sqrt{\frac{15}{8\pi}}\sin\theta\cos\theta e^{i\phi} \\
P_2^0(x) = & \tfrac{1}{2}(3x^2-1) & Y_{20} = & \sqrt{\frac{5}{4\pi}}(\tfrac{3}{2}\cos^2\theta - \tfrac{1}{2})
\end{array} \tag{6.7.5}$$

There are many bad ways to evaluate associated Legendre polynomials numerically. For example, there are explicit expressions, such as

$$P_l^m(x) = \frac{(-1)^m(l+m)!}{2^m m!(l-m)!}(1-x^2)^{m/2}\left[1 - \frac{(l-m)(m+l+1)}{1!(m+1)}\left(\frac{1-x}{2}\right) + \frac{(l-m)(l-m-1)(m+l+1)(m+l+2)}{2!(m+1)(m+2)}\left(\frac{1-x}{2}\right)^2 - \cdots\right] \tag{6.7.6}$$

where the polynomial continues up through the term in $(1-x)^{l-m}$. (See [1] for this and related formulas.) This is not a satisfactory method because evaluation of the polynomial involves delicate cancellations between successive terms, which alternate in sign. For large l, the individual terms in the polynomial become very much larger than their sum, and all accuracy is lost.

In practice, (6.7.6) can be used only in single precision (32-bit) for l up to 6 or 8, and in double precision (64-bit) for l up to 15 or 18, depending on the precision required for the answer. A more robust computational procedure is therefore desirable, as follows.

The associated Legendre functions satisfy numerous recurrence relations, tabulated in [1,2]. These are recurrences on l alone, on m alone, and on both l and m simultaneously. Most of the recurrences involving m are unstable, and so are dangerous for numerical work. The following recurrence on l is, however, stable (compare 5.4.1):

$$(l-m)P_l^m = x(2l-1)P_{l-1}^m - (l+m-1)P_{l-2}^m \tag{6.7.7}$$

Even this recurrence is useful only for moderate l and m, since the P_l^m's themselves grow rapidly with l and quickly overflow. The spherical harmonics by contrast remain bounded — after all, they are normalized to unity (eq. 6.7.1). It is exactly the square-root factor in equation (6.7.2) that balances the divergence. So the right function to use in the recurrence relation is the renormalized associated Legendre function,

$$\tilde{P}_l^m = \sqrt{\frac{2l+1}{4\pi}\frac{(l-m)!}{(l+m)!}}P_l^m \tag{6.7.8}$$

Then the recurrence relation (6.7.7) becomes

$$\tilde{P}_l^m = \sqrt{\frac{4l^2-1}{l^2-m^2}}\left[x\tilde{P}_{l-1}^m - \sqrt{\frac{(l-1)^2-m^2}{4(l-1)^2-1}}\tilde{P}_{l-2}^m\right] \tag{6.7.9}$$

We start the recurrence with the closed-form expression for the $l = m$ function,

$$\tilde{P}_m^m = (-1)^m\sqrt{\frac{2m+1}{4\pi(2m)!}}(2m-1)!!\,(1-x^2)^{m/2} \tag{6.7.10}$$

(The notation $n!!$ denotes the product of all *odd* integers less than or equal to n.) Using (6.7.9) with $l = m+1$, and setting $\tilde{P}_{m-1}^m = 0$, we find

$$\tilde{P}_{m+1}^m = x\sqrt{2m+3}\,\tilde{P}_m^m \tag{6.7.11}$$

Equations (6.7.10) and (6.7.11) provide the two starting values required for (6.7.9) for general l.

The function that implements this is

plegendre.h

```
Doub plegendre(const Int l, const Int m, const Doub x) {
```

Computes the renormalized associated Legendre polynomial $\tilde{P}_l^m(x)$, equation (6.7.8). Here m and l are integers satisfying $0 \le m \le l$, while x lies in the range $-1 \le x \le 1$.

```
    static const Doub PI=3.141592653589793;
    Int i,ll;
    Doub fact,oldfact,pll,pmm,pmmp1,omx2;
    if (m < 0 || m > l || abs(x) > 1.0)
        throw("Bad arguments in routine plgndr");
    pmm=1.0;                          Compute P̃_m^m.
    if (m > 0) {
        omx2=(1.0-x)*(1.0+x);
        fact=1.0;
        for (i=1;i<=m;i++) {
            pmm *= omx2*fact/(fact+1.0);
            fact += 2.0;
        }
    }
    pmm=sqrt((2*m+1)*pmm/(4.0*PI));
```

```
    if (m & 1)
        pmm=-pmm;
    if (l == m)
        return pmm;
    else {                          Compute P̃^m_{m+1}.
        pmmp1=x*sqrt(2.0*m+3.0)*pmm;
        if (l == (m+1))
            return pmmp1;
        else {                      Compute P̃^m_l, l > m + 1.
            oldfact=sqrt(2.0*m+3.0);
            for (ll=m+2;ll<=l;ll++) {
                fact=sqrt((4.0*ll*ll-1.0)/(ll*ll-m*m));
                pll=(x*pmmp1-pmm/oldfact)*fact;
                oldfact=fact;
                pmm=pmmp1;
                pmmp1=pll;
            }
            return pll;
        }
    }
}
```

Sometimes it is convenient to have the functions with the standard normalization, as defined by equation (6.7.4). Here is a routine that does this. Note that it will overflow for $m \gtrsim 80$, or even sooner if $l \gg m$.

plegendre.h

```
Doub plgndr(const Int l, const Int m, const Doub x)
```
Computes the associated Legendre polynomial $P_l^m(x)$, equation (6.7.4). Here m and l are integers satisfying $0 \le m \le l$, while x lies in the range $-1 \le x \le 1$. These functions will overflow for $m \gtrsim 80$.

```
{
    const Doub PI=3.141592653589793238;
    if (m < 0 || m > l || abs(x) > 1.0)
        throw("Bad arguments in routine plgndr");
    Doub prod=1.0;
    for (Int j=l-m+1;j<=l+m;j++)
        prod *= j;
    return sqrt(4.0*PI*prod/(2*l+1))*plegendre(l,m,x);
}
```

6.7.1 Fast Spherical Harmonic Transforms

Any smooth function on the surface of a sphere can be written as an expansion in spherical harmonics. Suppose the function can be well-approximated by truncating the expansion at $l = l_{\max}$.

$$\begin{aligned} f(\theta_i, \phi_j) &= \sum_{l=0}^{l_{\max}} \sum_{m=-l}^{m=l} a_{lm} Y_{lm}(\theta_i, \phi_j) \\ &= \sum_{l=0}^{l_{\max}} \sum_{m=-l}^{m=l} a_{lm} \tilde{P}_l^m(\cos\theta_i) e^{im\phi_j} \end{aligned} \tag{6.7.12}$$

Here we have written the function evaluated at one of N_θ sample points θ_i and one of N_ϕ sample points ϕ_j. The total number of sample points is $N = N_\theta N_\phi$. In applications, typically $N_\theta \sim N_\phi \sim \sqrt{N}$. Since the total number of spherical harmonics in the sum (6.7.12) is $l_{\max}^2$, we also have $l_{\max} \sim \sqrt{N}$.

How many operations does it take to evaluate the sum (6.7.12)? Direct evaluation of $l_{\max}^2$ terms at N sample points is an $O(N^2)$ process. You might try to speed this up by choosing the sample points ϕ_j to be equally spaced in angle and doing the sum over m by an FFT. Each FFT is $O(N_\phi \ln N_\phi)$, and you have to do $O(N_\theta l_{\max})$ of them, for a total of $O(N^{3/2} \ln N)$ operations, which is some improvement. A simple rearrangement [3-5] gives an even better way: Interchange the order of summation

$$\sum_{l=0}^{l_{\max}} \sum_{m=-l}^{l} \longleftrightarrow \sum_{m=-l_{\max}}^{l_{\max}} \sum_{l=|m|}^{l_{\max}} \tag{6.7.13}$$

so that

$$f(\theta_i, \phi_j) = \sum_{m=-l_{\max}}^{l_{\max}} q_m(\theta_i) e^{im\phi_j} \tag{6.7.14}$$

where

$$q_m(\theta_i) = \sum_{l=|m|}^{l_{\max}} a_{lm} \tilde{P}_l^m(\cos\theta_i) \tag{6.7.15}$$

Evaluating the sum in (6.7.15) is $O(l_{\max})$, and one must do this for $O(l_{\max} N_\theta)$ q_m's, so the total work is $O(N^{3/2})$. To evaluate equation (6.7.14) by an FFT at fixed θ_i is $O(N_\phi \ln N_\phi)$. There are N_θ FFTs to be done, for a total operations count of $O(N \ln N)$, which is negligible in comparison. So the total algorithm is $O(N^{3/2})$. Note that you can evaluate equation (6.7.14) either by precomputing and storing the $\tilde{P}_l^m$'s using the recurrence relation (6.7.9), or by Clenshaw's method (§5.4).

What about inverting the transform (6.7.12)? The formal inverse for the expansion of a continuous function $f(\theta, \phi)$ follows from the orthonormality of the Y_{lm}'s, equation (6.7.1),

$$a_{lm} = \int \sin\theta \, d\theta \, d\phi \; f(\theta, \phi) e^{-im\phi} \tilde{P}_l^m(\cos\theta) \tag{6.7.16}$$

For the discrete case, where we have a sampled function, the integral becomes a quadrature:

$$a_{lm} = \sum_{i,j} w(\theta_i) f(\theta_i, \phi_j) e^{-im\phi_j} \tilde{P}_l^m(\cos\theta_i) \tag{6.7.17}$$

Here $w(\theta_i)$ are the quadrature weights. In principle we could consider weights that depend on ϕ_j as well, but in practice we do the ϕ quadrature by an FFT, so the weights are unity. A good choice for the weights for an equi-angular grid in θ is given in Ref. [3], Theorem 3. Another possibility is to use Gaussian quadrature for the θ integral. In this case, you choose the sample points so that the $\cos\theta_i$'s are the abscissas returned by `gauleg` and the $w(\theta_i)$'s are the corresponding weights. The best way to organize the calculation is to first do the FFTs, computing

$$g_m(\theta_i) = \sum_j f(\theta_i, \phi_j) e^{-im\phi_j} \tag{6.7.18}$$

Then

$$a_{lm} = \sum_i w(\theta_i) g_m(\theta_i) \tilde{P}_l^m(\cos\theta_i) \tag{6.7.19}$$

You can verify that the operations count is dominated by equation (6.7.19) and scales as $O(N^{3/2})$ once again. In a real calculation, you should exploit all the symmetries that let you reduce the workload, such as $g_{-m} = g_m^*$ and $\tilde{P}_l^m[\cos(\pi - \theta)] = (-1)^{l+m}\tilde{P}_l^m(\cos\theta)$.

Very recently, algorithms for fast Legendre transforms have been developed, similar in spirit to the FFT [3,6,7]. Theoretically, they reduce the forward and inverse spherical harmonic transforms to $O(N \log^2 N)$ problems. However, current implementations [8] are not much faster than the $O(N^{3/2})$ methods above for $N \sim 500$, and there are stability and accuracy problems that require careful attention [9]. Stay tuned!

CITED REFERENCES AND FURTHER READING:

Magnus, W., and Oberhettinger, F. 1949, *Formulas and Theorems for the Functions of Mathematical Physics* (New York: Chelsea), pp. 54ff.[1]

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at `http://www.nr.com/aands`, Chapter 8.[2]

Driscoll, J.R., and Healy, D.M. 1994, "Computing Fourier Transforms and Convolutions on the 2–sphere," *Advances in Applied Mathematics*, vol. 15, pp. 202–250.[3]

Muciaccia, P.F., Natoli, P., and Vittorio, N. 1997, "Fast Spherical Harmonic Analysis: A Quick Algorithm for Generating and/or Inverting Full-Sky, High-Resolution Cosmic Microwave Background Anisotropy Maps," *Astrophysical Journal*, vol. 488, pp. L63–66.[4]

Oh, S.P., Spergel, D.N., and Hinshaw, G. 1999, "An Efficient Technique to Determine the Power Spectrum from Cosmic Microwave Background Sky Maps," *Astrophysical Journal*, vol. 510, pp. 551–563, Appendix A.[5]

Healy, D.M., Rockmore, D., Kostelec, P.J., and Moore, S. 2003, "FFTs for the 2-Sphere: Improvements and Variations," *Journal of Fourier Analysis and Applications*, vol. 9, pp. 341–385.[6]

Potts, D., Steidl, G., and Tasche, M. 1998, " Fast and Stable Algorithms for Discrete Spherical Fourier Transforms," *Linear Algebra and Its Applications*, vol. 275-276, pp. 433–450.[7]

Moore, S., Healy, D.M., Rockmore, D., and Kostelec, P.J. 2007+, *SpharmonicKit*. Software at `http://www.cs.dartmouth.edu/~geelong/sphere`.[8]

Healy, D.M., Kostelec, P.J., and Rockmore, D. 2004, "Towards Safe and Effective High-Order Legendre Transforms with Applications to FFTs for the 2-Sphere," *Advances in Computational Mathematics*, vol. 21, pp. 59–105.[9]

6.8 Fresnel Integrals, Cosine and Sine Integrals

6.8.1 Fresnel Integrals

The two Fresnel integrals are defined by

$$C(x) = \int_0^x \cos\left(\frac{\pi}{2}t^2\right) dt, \qquad S(x) = \int_0^x \sin\left(\frac{\pi}{2}t^2\right) dt \tag{6.8.1}$$

and are plotted in Figure 6.8.1.

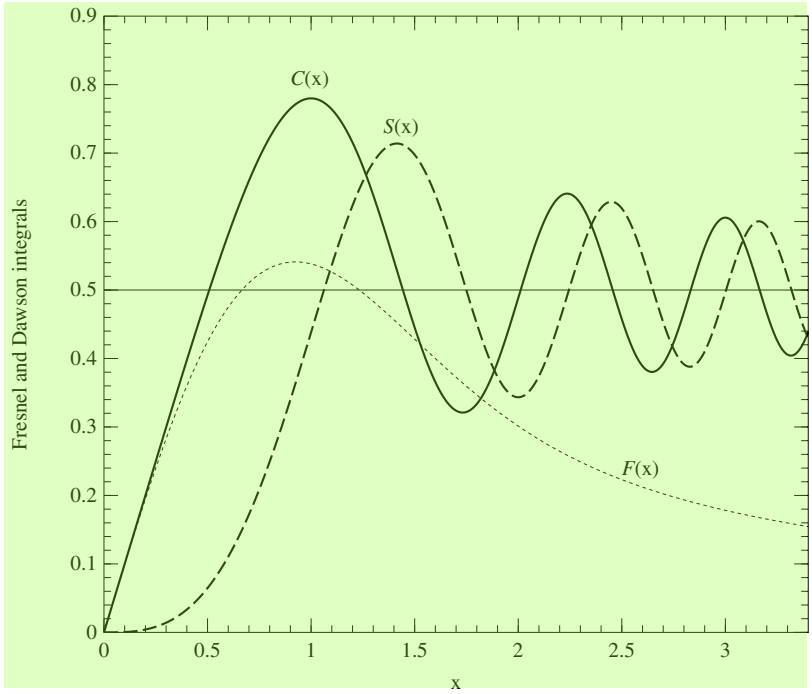

Figure 6.8.1. Fresnel integrals $C(x)$ and $S(x)$ (§6.8), and Dawson's integral $F(x)$ (§6.9).

The most convenient way of evaluating these functions to arbitrary precision is to use power series for small x and a continued fraction for large x. The series are

$$C(x) = x - \left(\frac{\pi}{2}\right)^2 \frac{x^5}{5 \cdot 2!} + \left(\frac{\pi}{2}\right)^4 \frac{x^9}{9 \cdot 4!} - \cdots$$
$$S(x) = \left(\frac{\pi}{2}\right) \frac{x^3}{3 \cdot 1!} - \left(\frac{\pi}{2}\right)^3 \frac{x^7}{7 \cdot 3!} + \left(\frac{\pi}{2}\right)^5 \frac{x^{11}}{11 \cdot 5!} - \cdots \tag{6.8.2}$$

There is a complex continued fraction that yields both $S(x)$ and $C(x)$ simultaneously:

$$C(x) + iS(x) = \frac{1+i}{2} \operatorname{erf} z, \qquad z = \frac{\sqrt{\pi}}{2}(1-i)x \tag{6.8.3}$$

where

$$e^{z^2} \operatorname{erfc} z = \frac{1}{\sqrt{\pi}} \left(\frac{1}{z+} \frac{1/2}{z+} \frac{1}{z+} \frac{3/2}{z+} \frac{2}{z+} \cdots \right)$$
$$= \frac{2z}{\sqrt{\pi}} \left(\frac{1}{2z^2+1-} \frac{1 \cdot 2}{2z^2+5-} \frac{3 \cdot 4}{2z^2+9-} \cdots \right) \tag{6.8.4}$$

In the last line we have converted the "standard" form of the continued fraction to its "even" form (see §5.2), which converges twice as fast. We must be careful not to evaluate the alternating series (6.8.2) at too large a value of x; inspection of the terms shows that $x = 1.5$ is a good point to switch over to the continued fraction.

Note that for large x

$$C(x) \sim \frac{1}{2} + \frac{1}{\pi x} \sin\left(\frac{\pi}{2}x^2\right), \qquad S(x) \sim \frac{1}{2} - \frac{1}{\pi x} \cos\left(\frac{\pi}{2}x^2\right) \tag{6.8.5}$$

Thus the precision of the routine `frenel` may be limited by the precision of the library routines for sine and cosine for large x.

frenel.h

```
Complex frenel(const Doub x) {
```

Computes the Fresnel integrals $S(x)$ and $C(x)$ for all real x. $C(x)$ is returned as the real part of cs and $S(x)$ as the imaginary part.

```
    static const Int MAXIT=100;
    static const Doub PI=3.141592653589793238, PIBY2=(PI/2.0), XMIN=1.5,
        EPS=numeric_limits<Doub>::epsilon(),
        FPMIN=numeric_limits<Doub>::min(),
        BIG=numeric_limits<Doub>::max()*EPS;
```

Here `MAXIT` is the maximum number of iterations allowed; `EPS` is the relative error; `FPMIN` is a number near the smallest representable floating-point number; `BIG` is a number near the machine overflow limit; and `XMIN` is the dividing line between using the series and continued fraction.

```
    Bool odd;
    Int k,n;
    Doub a,ax,fact,pix2,sign,sum,sumc,sums,term,test;
    Complex b,cc,d,h,del,cs;
    if ((ax=abs(x)) < sqrt(FPMIN)) {
        cs=ax;
    } else if (ax <= XMIN) {
        sum=sums=0.0;
        sumc=ax;
        sign=1.0;
        fact=PIBY2*ax*ax;
        odd=true;
        term=ax;
        n=3;
        for (k=1;k<=MAXIT;k++) {
            term *= fact/k;
            sum += sign*term/n;
            test=abs(sum)*EPS;
            if (odd) {
                sign = -sign;
                sums=sum;
                sum=sumc;
            } else {
                sumc=sum;
                sum=sums;
            }
            if (term < test) break;
            odd=!odd;
            n += 2;
        }
        if (k > MAXIT) throw("series failed in frenel");
        cs=Complex(sumc,sums);
    } else {
        pix2=PI*ax*ax;
        b=Complex(1.0,-pix2);
        cc=BIG;
        d=h=1.0/b;
        n = -1;
        for (k=2;k<=MAXIT;k++) {
            n += 2;
            a = -n*(n+1);
            b += 4.0;
            d=1.0/(a*d+b);
            cc=b+a/cc;
            del=cc*d;
            h *= del;
            if (abs(real(del)-1.0)+abs(imag(del)) <= EPS) break;
        }
```

Special case: Avoid failure of convergence test because of underflow.

Evaluate both series simultaneously.

Evaluate continued fraction by modified Lentz's method (§5.2).

Denominators cannot be zero.

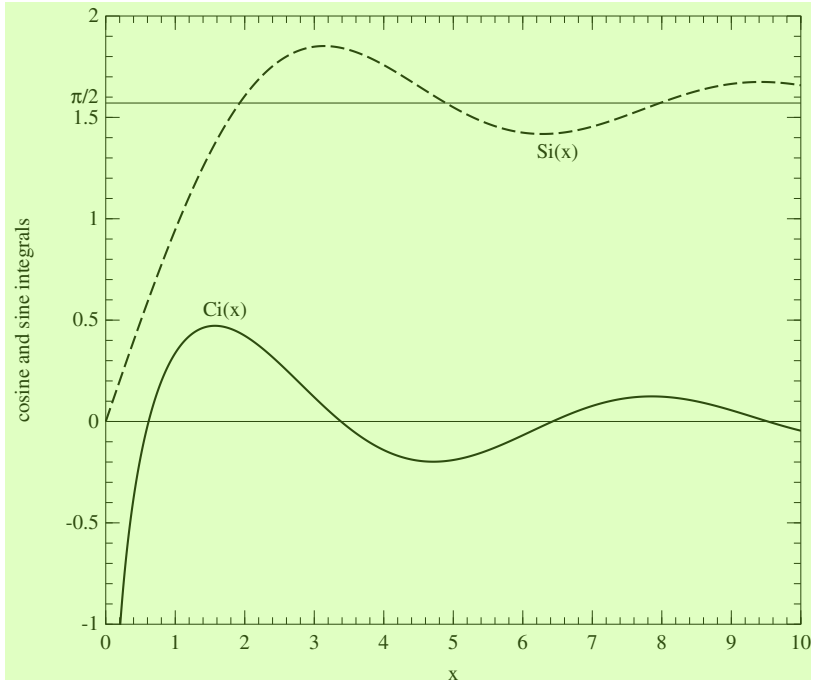

Figure 6.8.2. Sine and cosine integrals $\mathrm{Si}(x)$ and $\mathrm{Ci}(x)$.

```
        if (k > MAXIT) throw("cf failed in frenel");
        h *= Complex(ax,-ax);
        cs=Complex(0.5,0.5)
            *(1.0-Complex(cos(0.5*pix2),sin(0.5*pix2))*h);
    }
    if (x < 0.0) cs = -cs;                    Use antisymmetry.
    return cs;
}
```

6.8.2 Cosine and Sine Integrals

The cosine and sine integrals are defined by

$$\mathrm{Ci}(x) = \gamma + \ln x + \int_0^x \frac{\cos t - 1}{t}\,dt$$
$$\mathrm{Si}(x) = \int_0^x \frac{\sin t}{t}\,dt \qquad (6.8.6)$$

and are plotted in Figure 6.8.2. Here $\gamma \approx 0.5772$ is Euler's constant. We only need a way to calculate the functions for $x > 0$, because

$$\mathrm{Si}(-x) = -\mathrm{Si}(x), \qquad \mathrm{Ci}(-x) = \mathrm{Ci}(x) - i\pi \qquad (6.8.7)$$

Once again we can evaluate these functions by a judicious combination of power

series and complex continued fraction. The series are

$$\begin{aligned}\mathrm{Si}(x) &= x - \frac{x^3}{3\cdot 3!} + \frac{x^5}{5\cdot 5!} - \cdots \\ \mathrm{Ci}(x) &= \gamma + \ln x + \left(-\frac{x^2}{2\cdot 2!} + \frac{x^4}{4\cdot 4!} - \cdots\right)\end{aligned} \tag{6.8.8}$$

The continued fraction for the exponential integral $E_1(ix)$ is

$$\begin{aligned}E_1(ix) &= -\mathrm{Ci}(x) + i[\mathrm{Si}(x) - \pi/2] \\ &= e^{-ix}\left(\frac{1}{ix+}\,\frac{1}{1+}\,\frac{1}{ix+}\,\frac{2}{1+}\,\frac{2}{ix+}\cdots\right) \\ &= e^{-ix}\left(\frac{1}{1+ix-}\,\frac{1^2}{3+ix-}\,\frac{2^2}{5+ix-}\cdots\right)\end{aligned} \tag{6.8.9}$$

The "even" form of the continued fraction is given in the last line and converges twice as fast for about the same amount of computation. A good crossover point from the alternating series to the continued fraction is $x = 2$ in this case. As for the Fresnel integrals, for large x the precision may be limited by the precision of the sine and cosine routines.

```
Complex cisi(const Doub x) {
```
cisi.h

Computes the cosine and sine integrals $\mathrm{Ci}(x)$ and $\mathrm{Si}(x)$. The function $\mathrm{Ci}(x)$ is returned as the real part of cs, and $\mathrm{Si}(x)$ as the imaginary part. $\mathrm{Ci}(0)$ is returned as a large negative number and no error message is generated. For $x < 0$ the routine returns $\mathrm{Ci}(-x)$ and you must supply the $-i\pi$ yourself.

```
    static const Int MAXIT=100;                    Maximum number of iterations allowed.
    static const Doub EULER=0.577215664901533, PIBY2=1.570796326794897,
        TMIN=2.0, EPS=numeric_limits<Doub>::epsilon(),
        FPMIN=numeric_limits<Doub>::min()*4.0,
        BIG=numeric_limits<Doub>::max()*EPS;
```

Here `EULER` is Euler's constant γ; `PIBY2` is $\pi/2$; `TMIN` is the dividing line between using the series and continued fraction; `EPS` is the relative error, or absolute error near a zero of $\mathrm{Ci}(x)$; `FPMIN` is a number close to the smallest representable floating-point number; and `BIG` is a number near the machine overflow limit.

```
    Int i,k;
    Bool odd;
    Doub a,err,fact,sign,sum,sumc,sums,t,term;
    Complex h,b,c,d,del,cs;
    if ((t=abs(x)) == 0.0) return -BIG;            Special case.
    if (t > TMIN) {                                Evaluate continued fraction by modified
        b=Complex(1.0,t);                              Lentz's method (§5.2).
        c=Complex(BIG,0.0);
        d=h=1.0/b;
        for (i=1;i<MAXIT;i++) {
            a= -i*i;
            b += 2.0;
            d=1.0/(a*d+b);                         Denominators cannot be zero.
            c=b+a/c;
            del=c*d;
            h *= del;
            if (abs(real(del)-1.0)+abs(imag(del)) <= EPS) break;
        }
        if (i >= MAXIT) throw("cf failed in cisi");
        h=Complex(cos(t),-sin(t))*h;
        cs= -conj(h)+Complex(0.0,PIBY2);
    } else {                                       Evaluate both series simultaneously.
```

```
        if (t < sqrt(FPMIN)) {                    Special case: Avoid failure of convergence test because of underflow.
            sumc=0.0;
            sums=t;
        } else {
            sum=sums=sumc=0.0;
            sign=fact=1.0;
            odd=true;
            for (k=1;k<=MAXIT;k++) {
                fact *= t/k;
                term=fact/k;
                sum += sign*term;
                err=term/abs(sum);
                if (odd) {
                    sign = -sign;
                    sums=sum;
                    sum=sumc;
                } else {
                    sumc=sum;
                    sum=sums;
                }
                if (err < EPS) break;
                odd=!odd;
            }
            if (k > MAXIT) throw("maxits exceeded in cisi");
        }
        cs=Complex(sumc+log(t)+EULER,sums);
    }
    if (x < 0.0) cs = conj(cs);
    return cs;
}
```


CITED REFERENCES AND FURTHER READING:

Stegun, I.A., and Zucker, R. 1976, "Automatic Computing Methods for Special Functions. III. The Sine, Cosine, Exponential integrals, and Related Functions," *Journal of Research of the National Bureau of Standards*, vol. 80B, pp. 291–311; 1981, "Automatic Computing Methods for Special Functions. IV. Complex Error Function, Fresnel Integrals, and Other Related Functions," *op. cit.*, vol. 86, pp. 661–686.

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at `http://www.nr.com/aands`, Chapters 5 and 7.

6.9 Dawson's Integral

Dawson's Integral $F(x)$ is defined by

$$F(x) = e^{-x^2} \int_0^x e^{t^2}\, dt \tag{6.9.1}$$

See Figure 6.8.1 for a graph of the function. The function can also be related to the complex error function by

$$F(z) = \frac{i\sqrt{\pi}}{2} e^{-z^2} [1 - \mathrm{erfc}(-iz)]. \tag{6.9.2}$$

A remarkable approximation for $F(z)$ due to Rybicki [1], is

$$F(z) = \lim_{h\to 0} \frac{1}{\sqrt{\pi}} \sum_{n \text{ odd}} \frac{e^{-(z-nh)^2}}{n} \tag{6.9.3}$$

What makes equation (6.9.3) unusual is that its accuracy increases *exponentially* as h gets small, so that quite moderate values of h (and correspondingly quite rapid convergence of the series) give very accurate approximations.

We will discuss the theory that leads to equation (6.9.3) later, in §13.11, as an interesting application of Fourier methods. Here we simply implement a routine for real values of x based on the formula.

It is first convenient to shift the summation index to center it approximately on the maximum of the exponential term. Define n_0 to be the even integer nearest to x/h, and $x_0 \equiv n_0 h$, $x' \equiv x - x_0$, and $n' \equiv n - n_0$, so that

$$F(x) \approx \frac{1}{\sqrt{\pi}} \sum_{n'=-N \text{ odd}}^{N} \frac{e^{-(x'-n'h)^2}}{n'+n_0} \tag{6.9.4}$$

where the approximate equality is accurate when h is sufficiently small and N is sufficiently large. The computation of this formula can be greatly speeded up if we note that

$$e^{-(x'-n'h)^2} = e^{-x'^2} e^{-(n'h)^2} \left(e^{2x'h}\right)^{n'} \tag{6.9.5}$$

The first factor is computed once, the second is an array of constants to be stored, and the third can be computed recursively, so that only two exponentials need be evaluated. Advantage is also taken of the symmetry of the coefficients $e^{-(n'h)^2}$ by breaking up the summation into positive and negative values of n' separately.

In the following routine, the choices $h = 0.4$ and $N = 11$ are made. Because of the symmetry of the summations and the restriction to odd values of n, the limits on the `for` loops are 0 to 5. The accuracy of the result in this version is about 2×10^{-7}. In order to maintain relative accuracy near $x = 0$, where $F(x)$ vanishes, the program branches to the evaluation of the power series [2] for $F(x)$, for $|x| < 0.2$.

dawson.h

```
Doub dawson(const Doub x) {
Returns Dawson's integral F(x) = exp(-x^2) ∫_0^x exp(t^2)dt for any real x.
    static const Int NMAX=6;
    static VecDoub c(NMAX);
    static Bool init = true;
    static const Doub H=0.4, A1=2.0/3.0, A2=0.4, A3=2.0/7.0;
    Int i,n0;                          Flag is true if we need to initialize, else false.
    Doub d1,d2,e1,e2,sum,x2,xp,xx,ans;
    if (init) {
        init=false;
        for (i=0;i<NMAX;i++) c[i]=exp(-SQR((2.0*i+1.0)*H));
    }
    if (abs(x) < 0.2) {                Use series expansion.
        x2=x*x;
        ans=x*(1.0-A1*x2*(1.0-A2*x2*(1.0-A3*x2)));
    } else {                           Use sampling theorem representation.
        xx=abs(x);
        n0=2*Int(0.5*xx/H+0.5);
        xp=xx-n0*H;
        e1=exp(2.0*xp*H);
```

```
        e2=e1*e1;
        d1=n0+1;
        d2=d1-2.0;
        sum=0.0;
        for (i=0;i<NMAX;i++,d1+=2.0,d2-=2.0,e1*=e2)
            sum += c[i]*(e1/d1+1.0/(d2*e1));
        ans=0.5641895835*SIGN(exp(-xp*xp),x)*sum;
    }
    return ans;
}
```

Constant is $1/\sqrt{\pi}$.

Other methods for computing Dawson's integral are also known [2,3].

CITED REFERENCES AND FURTHER READING:

Rybicki, G.B. 1989, "Dawson's Integral and The Sampling Theorem," *Computers in Physics*, vol. 3, no. 2, pp. 85–87.[1]

Cody, W.J., Pociorek, K.A., and Thatcher, H.C. 1970, "Chebyshev Approximations for Dawson's Integral," *Mathematics of Computation*, vol. 24, pp. 171–178.[2]

McCabe, J.H. 1974, "A Continued Fraction Expansion, with a Truncation Error Estimate, for Dawson's Integral," *Mathematics of Computation*, vol. 28, pp. 811–816.[3]

6.10 Generalized Fermi-Dirac Integrals

The generalized Fermi-Dirac integral is defined as

$$F_k(\eta,\theta) = \int_0^\infty \frac{x^k(1+\frac{1}{2}\theta x)^{1/2}}{e^{x-\eta}+1}\,dx \tag{6.10.1}$$

It occurs, for example, in astrophysical applications with θ nonnegative and arbitrary η. In condensed matter physics one usually has the simpler case of $\theta = 0$ and omits the "generalized" from the name of the function. The important values of k are $-1/2$, $1/2$, $3/2$, and $5/2$, but we'll consider arbitrary values greater than -1. Watch out for an alternative definition that multiplies the integral by $1/\Gamma(k+1)$.

For $\eta \ll -1$ and $\eta \gg 1$ there are useful series expansions for these functions (see, e.g., [1]). These give, for example,

$$\begin{aligned} F_{1/2}(\eta,\theta) &\to \frac{1}{\sqrt{2\theta}} e^\eta e^{1/\theta} K_1\left(\frac{1}{\theta}\right), \quad \eta \to -\infty \\ F_{1/2}(\eta,\theta) &\to \frac{1}{2\sqrt{2}}\eta^{3/2}\frac{y\sqrt{1+y^2}-\sinh^{-1}y}{(\sqrt{1+y^2}-1)^{3/2}}, \quad \eta \to \infty \end{aligned} \tag{6.10.2}$$

Here y is defined by

$$1+y^2 = (1+\eta\theta)^2 \tag{6.10.3}$$

It is the middle range of η values that is difficult to handle.

For $\theta = 0$, Macleod [2] has given Chebyshev expansions accurate to 10^{-16} for the four important k values, covering all η values. In this case, one need look no further for an algorithm. Goano [3] handles arbitrary k for $\theta = 0$. For nonzero θ,

it is reasonable to compute the functions by direct integration, using variable transformation to get rapidly converging quadratures [4]. (Of course, this works also for $\theta = 0$, but is not as efficient.) The usual transformation $x = \exp(t - e^{-t})$ handles the singularity at $x = 0$ and the exponential fall off at large x (cf. equation 4.5.14). For $\eta \gtrsim 15$, it is better to split the integral into two regions, $[0, \eta]$ and $[\eta, \eta + 60]$. (The contribution beyond $\eta + 60$ is negligible.) Each of these integrals can then be done with the DE rule. Between 60 and 500 function evaluations give full double precision, larger η requiring more function evaluations. A more efficient strategy would replace the quadrature by a series expansion for large η.

In the implementation below, note how `operator()` is overloaded to define both a function of one variable (for `Trapzd`) and a function of two variables (for `DErule`). Note also the syntax

```
Trapzd<Fermi> s(*this,a,b);
```

for declaring a `Trapzd` object inside the `Fermi` object itself.

fermi.h

```
struct Fermi {
    Doub kk,etaa,thetaa;
    Doub operator() (const Doub t);
    Doub operator() (const Doub x, const Doub del);
    Doub val(const Doub k, const Doub eta, const Doub theta);
};

Doub Fermi::operator() (const Doub t) {
```

Integrand for trapezoidal quadrature of generalized Fermi-Dirac integral with transformation $x = \exp(t - e^{-t})$.

```
    Doub x;
    x=exp(t-exp(-t));
    return x*(1.0+exp(-t))*pow(x,kk)*sqrt(1.0+thetaa*0.5*x)/
        (exp(x-etaa)+1.0);
}

Doub Fermi::operator() (const Doub x, const Doub del) {
```

Integrand for DE rule quadrature of generalized Fermi-Dirac integral.

```
    if (x < 1.0)
        return pow(del,kk)*sqrt(1.0+thetaa*0.5*x)/(exp(x-etaa)+1.0);
    else
        return pow(x,kk)*sqrt(1.0+thetaa*0.5*x)/(exp(x-etaa)+1.0);
}

Doub Fermi::val(const Doub k, const Doub eta, const Doub theta)
```

Computes the generalized Fermi-Dirac integral $F_k(\eta, \theta)$, where $k > -1$ and $\theta \geq 0$. The accuracy is approximately the square of the parameter `EPS`. `NMAX` limits the total number of quadrature steps.

```
{
    const Doub EPS=3.0e-9;
    const Int NMAX=11;
    Doub a,aa,b,bb,hmax,olds,sum;
    kk=k;
    etaa=eta;
    thetaa=theta;
    if (eta <= 15.0) {
        a=-4.5;
        b=5.0;
        Trapzd<Fermi> s(*this,a,b);
        for (Int i=1;i<=NMAX;i++) {
            sum=s.next();
            if (i > 3)
                if (abs(sum-olds) <= EPS*abs(olds))
```

Load the arguments into the member variables for use in the function evaluations.

Set limits for $x = \exp(t - e^{-t})$ mapping.

Test for convergence.

```
                return sum;
            olds=sum;                    Save value for next convergence test.
        }
    }
    else {
        a=0.0;                           Set limits for DE rule.
        b=eta;
        aa=eta;
        bb=eta+60.0;
        hmax=4.3;                        Big enough to handle negative k or large η.
        DErule<Fermi> s(*this,a,b,hmax);
        DErule<Fermi> ss(*this,aa,bb,hmax);
        for (Int i=1;i<=NMAX;i++) {
            sum=s.next()+ss.next();
            if (i > 3)
                if (abs(sum-olds) <= EPS*abs(olds))
                    return sum;
            olds=sum;
        }
    }
    throw("no convergence in fermi");
    return 0.0;
}
```

You get values of the Fermi-Dirac functions by declaring a `Fermi` object:

```
Fermi ferm;
```

and then making repeated calls to the `val` function:

```
ans=ferm.val(k,eta,theta);
```

Other quadrature methods exist for these functions [5-7]. A reasonably efficient method [8] involves trapezoidal quadrature with "pole correction," but it is restricted to $\theta \lesssim 0.2$. Generalized Bose-Einstein integrals can also be computed by the DE rule or the methods in these references.

CITED REFERENCES AND FURTHER READING:

Cox, J.P., and Giuli, R.T. 1968, *Principles of Stellar Structure* (New York: Gordon and Breach), vol. II, §24.7.[1]

Macleod, A.J. 1998, "Fermi-Dirac Functions of Order $-1/2$, $1/2$, $3/2$, $5/2$," *ACM Transactions on Mathematical Software*, vol. 24, pp. 1–12. (Algorithm 779, available from netlib.)[2]

Goano, M. 1995, "Computation of the Complete and Incomplete Fermi-Dirac Integral," *ACM Transactions on Mathematical Software*, vol. 21, pp. 221–232. (Algorithm 745, available from netlib.)[3]

Natarajan, A., and Kumar, N.M. 1993, "On the Numerical Evaluation of the Generalised Fermi-Dirac Integrals," *Computer Physics Communications*, vol. 76, pp. 48–50.[4]

Pichon, B. 1989, "Numerical Calculation of the Generalized Fermi-Dirac Integrals," *Computer Physics Communications*, vol. 55, pp. 127–136.[5]

Sagar, R.P. 1991, "A Gaussian Quadrature for the Calculation of Generalized Fermi-Dirac Integrals," *Computer Physics Communications*, vol. 66, pp. 271–275.[6]

Gautschi, W. 1992, "On the Computation of Generalized Fermi-Dirac and Bose-Einstein Integrals," *Computer Physics Communications*, vol. 74, pp. 233–238.[7]

Mohankumar, N., and Natarajan, A. 1996, "A Note on the Evaluation of the Generalized Fermi-Dirac Integral," *Astrophysical Journal*, vol. 458, pp. 233–235.[8]

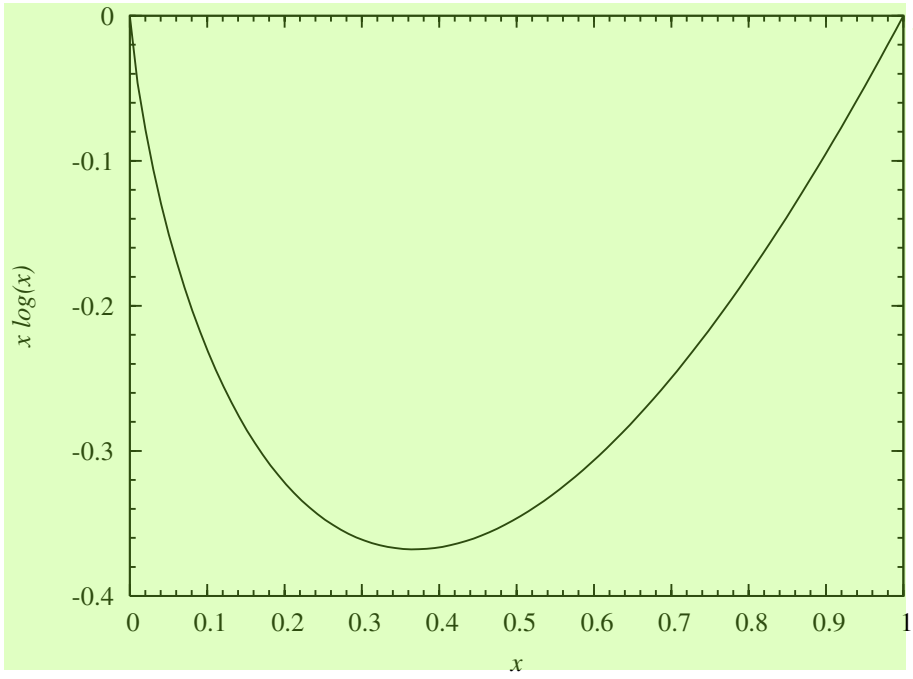

Figure 6.11.1. The function $x \log(x)$ is shown for $0 < x < 1$. Although nearly invisible, an essential singularity at $x = 0$ makes this function tricky to invert.

6.11 Inverse of the Function x log(x)

The function

$$y(x) = x \log(x) \tag{6.11.1}$$

and its inverse function $x(y)$ occur in a number of statistical and information theoretical contexts. Obviously $y(x)$ is nonsingular for all positive x, and easy to evaluate. For x between 0 and 1, it is negative, with a single minimum at $(x, y) = (e^{-1}, -e^{-1})$. The function has the value 0 at $x = 1$ and it has the value 0 as its limit at $x = 0$, since the linear factor x easily dominates the singular logarithm.

Computing the inverse function $x(y)$ is, however, not so easy. (We will need this inverse in §6.14.12.) From the appearance of Figure 6.11.1, it might seem easy to invert the function on its left branch, that is, return a value x between 0 and e^{-1} for every value y between 0 and $-e^{-1}$. However, the lurking logarithmic singularity at $x = 0$ causes difficulties for many methods that you might try.

Polynomial fits work well over any range of y that is less than a decade or so (e.g., from 0.01 to 0.1), but fail badly if you demand high fractional precision extending all the way to $y = 0$.

What about Newton's method? We write

$$\begin{aligned} f(x) &\equiv x \log(x) - y \\ f'(x) &= 1 + \log(x) \end{aligned} \tag{6.11.2}$$

giving the iteration

$$x_{i+1} = x_i - \frac{x_i \log(x_i) - y}{1 + \log(x_i)} \tag{6.11.3}$$

This doesn't work. The problem is not with its rate of convergence, which is of course quadratic for any finite y *if* we start close enough to the solution (see §9.4). The problem is that the region in which it converges at all is very small, especially as $y \to 0$. So, if we don't already have a good approximation as we approach the singularity, we are sunk.

If we change variables, we can get different (not computationally equivalent) versions of Newton's method. For example, let

$$u \equiv \log(x), \qquad x = e^u \tag{6.11.4}$$

Newton's method in u looks like this:

$$\begin{aligned} f(u) &= ue^u - y \\ f'(u) &= (1+u)e^u \\ u_{i+1} &= u_i - \frac{u_i - e^{-u_i} y}{1+u_i} \end{aligned} \tag{6.11.5}$$

But it turns out that iteration (6.11.5) is no better than (6.11.3).

The observation that leads to a good solution is that, since its log term varies only slowly, $y = x\log(x)$ is only very modestly curved *when it is plotted in log-log coordinates*. (Actually it is the negative of y that is plotted, since log-log coordinates require positive quantities.) Algebraically, we rewrite equation (6.11.1) as

$$(-y) = (-u)e^u \tag{6.11.6}$$

(with u as defined above) and take logarithms, giving

$$\log(-y) = u + \log(-u) \tag{6.11.7}$$

This leads to the Newton formulas,

$$\begin{aligned} f(u) &= u + \log(-u) - \log(-y) \\ f'(u) &= \frac{u+1}{u} \\ u_{i+1} &= u_i + \frac{u_i}{u_i+1}\left[\log\left(\frac{y}{u_i}\right) - u_i\right] \end{aligned} \tag{6.11.8}$$

It turns out that the iteration (6.11.8) converges quadratically over quite a broad region of initial guesses. For $-0.2 < y < 0$, you can just choose -10 (for example) as a fixed initial guess. When $-0.2 < y < -e^{-1}$, one can use the Taylor series expansion around $x = e^{-1}$,

$$y(x - e^{-1}) = -e^{-1} + \tfrac{1}{2}e(x - e^{-1})^2 + \cdots \tag{6.11.9}$$

which yields

$$x \approx e^{-1} - \sqrt{2e^{-1}(y + e^{-1})} \tag{6.11.10}$$

With these initial guesses, (6.11.8) never takes more than six iterations to converge to double precision accuracy, and there is just one log and a few arithmetic operations per iteration. The implementation looks like this:

```
Doub invxlogx(Doub y) {
For negative y, 0 > y > -e^-1, return x such that y = x log(x). The value returned is always
the smaller of the two roots and is in the range 0 < x < e^-1.
    const Doub ooe = 0.367879441171442322;
    Doub t,u,to=0.;
    if (y >= 0. || y <= -ooe) throw("no such inverse value");
     if (y < -0.2) u = log(ooe-sqrt(2*ooe*(y+ooe)));  First approximation by inverse
    else u = -10.;                                        of Taylor series.
    do {                                                  See text for derivation.
        u += (t=(log(y/u)-u)*(u/(1.+u)));
        if (t < 1.e-8 && abs(t+to)<0.01*abs(t)) break;
        to = t;
    } while (abs(t/u) > 1.e-15);
    return exp(u);
}
```

ksdist.h

6.12 Elliptic Integrals and Jacobian Elliptic Functions

Elliptic integrals occur in many applications, because any integral of the form

$$\int R(t,s)\,dt \tag{6.12.1}$$

where R is a rational function of t and s, and s is the square root of a cubic or quartic polynomial in t, can be evaluated in terms of elliptic integrals. Standard references [1] describe how to carry out the reduction, which was originally done by Legendre. Legendre showed that only three basic elliptic integrals are required. The simplest of these is

$$I_1 = \int_y^x \frac{dt}{\sqrt{(a_1+b_1t)(a_2+b_2t)(a_3+b_3t)(a_4+b_4t)}} \tag{6.12.2}$$

where we have written the quartic s^2 in factored form. In standard integral tables [2], one of the limits of integration is always a zero of the quartic, while the other limit lies closer than the next zero, so that there is no singularity within the interval. To evaluate I_1 we simply break the interval $[y,x]$ into subintervals, each of which either begins or ends on a singularity. The tables, therefore, need only distinguish the eight cases in which each of the four zeros (ordered according to size) appears as the upper or lower limit of integration. In addition, when one of the b's in (6.12.2) tends to zero, the quartic reduces to a cubic, with the largest or smallest singularity moving to $\pm\infty$; this leads to eight more cases (actually just special cases of the first eight). The 16 cases in total are then usually tabulated in terms of Legendre's standard elliptic integral of the first kind, which we will define below. By a change of the variable of integration t, the zeros of the quartic are mapped to standard locations on the real axis. Then only two dimensionless parameters are needed to tabulate Legendre's integral. However, the symmetry of the original integral (6.12.2) under permutation of the roots is concealed in Legendre's notation. We will get back to Legendre's notation below. But first, here is a better approach:

Carlson [3] has given a new definition of a standard elliptic integral of the first kind,

$$R_F(x,y,z) = \frac{1}{2}\int_0^\infty \frac{dt}{\sqrt{(t+x)(t+y)(t+z)}} \tag{6.12.3}$$

where x, y, and z are nonnegative and at most one is zero. By standardizing the range of integration, he retains permutation symmetry for the zeros. (Weierstrass' canonical form also has this property.) Carlson first shows that when x or y is a zero of the quartic in (6.12.2), the integral I_1 can be written in terms of R_F in a form that is symmetric under permutation of the *remaining* three zeros. In the general case, when neither x nor y is a zero, two such R_F functions can be combined into a single one by an *addition theorem*, leading to the fundamental formula

$$I_1 = 2R_F(U_{12}^2, U_{13}^2, U_{14}^2) \tag{6.12.4}$$

where

$$U_{ij} = (X_i X_j Y_k Y_m + Y_i Y_j X_k X_m)/(x-y) \tag{6.12.5}$$

$$X_i = (a_i + b_i x)^{1/2}, \qquad Y_i = (a_i + b_i y)^{1/2} \tag{6.12.6}$$

and i, j, k, m is any permutation of $1, 2, 3, 4$. A short-cut in evaluating these expressions is

$$\begin{aligned} U_{13}^2 &= U_{12}^2 - (a_1 b_4 - a_4 b_1)(a_2 b_3 - a_3 b_2) \\ U_{14}^2 &= U_{12}^2 - (a_1 b_3 - a_3 b_1)(a_2 b_4 - a_4 b_2) \end{aligned} \tag{6.12.7}$$

The U's correspond to the three ways of pairing the four zeros, and I_1 is thus manifestly symmetric under permutation of the zeros. Equation (6.12.4) therefore reproduces all 16 cases when one limit is a zero, and also includes the cases when neither limit is a zero.

Thus Carlson's function allows arbitrary ranges of integration and arbitrary positions of the branch points of the integrand relative to the interval of integration. To handle elliptic integrals of the second and third kinds, Carlson defines the standard integral of the third kind as

$$R_J(x,y,z,p) = \frac{3}{2}\int_0^\infty \frac{dt}{(t+p)\sqrt{(t+x)(t+y)(t+z)}} \tag{6.12.8}$$

which is symmetric in x, y, and z. The degenerate case when two arguments are equal is denoted

$$R_D(x,y,z) = R_J(x,y,z,z) \tag{6.12.9}$$

and is symmetric in x and y. The function R_D replaces Legendre's integral of the second kind. The degenerate form of R_F is denoted

$$R_C(x,y) = R_F(x,y,y) \tag{6.12.10}$$

It embraces logarithmic, inverse circular, and inverse hyperbolic functions.

Carlson [4-7] gives integral tables in terms of the exponents of the linear factors of the quartic in (6.12.1). For example, the integral where the exponents are $(\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, -\frac{3}{2})$ can be expressed as a single integral in terms of R_D; it accounts for 144 separate cases in Gradshteyn and Ryzhik [2]!

Refer to Carlson's papers [3-8] for some of the practical details in reducing elliptic integrals to his standard forms, such as handling complex-conjugate zeros.

Turn now to the numerical evaluation of elliptic integrals. The traditional methods [9] are Gauss or Landen transformations. *Descending* transformations decrease the modulus k of the Legendre integrals toward zero, and *increasing* transformations increase it toward unity. In these limits the functions have simple analytic expressions. While these methods converge quadratically and are quite satisfactory for integrals of the first and second kinds, they generally lead to loss of significant figures in certain regimes for integrals of the third kind. Carlson's algorithms [10,11], by contrast, provide a unified method for all three kinds with no significant cancellations.

The key ingredient in these algorithms is the *duplication theorem*:

$$R_F(x,y,z) = 2R_F(x+\lambda, y+\lambda, z+\lambda) = R_F\left(\frac{x+\lambda}{4}, \frac{y+\lambda}{4}, \frac{z+\lambda}{4}\right) \tag{6.12.11}$$

where

$$\lambda = (xy)^{1/2} + (xz)^{1/2} + (yz)^{1/2} \tag{6.12.12}$$

This theorem can be proved by a simple change of variable of integration [12]. Equation (6.12.11) is iterated until the arguments of R_F are nearly equal. For equal arguments we have

$$R_F(x,x,x) = x^{-1/2} \tag{6.12.13}$$

When the arguments are close enough, the function is evaluated from a fixed Taylor expansion about (6.12.13) through fifth-order terms. While the iterative part of the algorithm is only linearly convergent, the error ultimately decreases by a factor of $4^6 = 4096$ for each iteration. Typically only two or three iterations are required, perhaps six or seven if the initial values of the arguments have huge ratios. We list the algorithm for R_F here, and refer you to Carlson's paper [10] for the other cases.

Stage 1: For $n = 0, 1, 2, \ldots$, compute

$$\mu_n = (x_n + y_n + z_n)/3$$
$$X_n = 1 - (x_n/\mu_n), \quad Y_n = 1 - (y_n/\mu_n), \quad Z_n = 1 - (z_n/\mu_n)$$
$$\epsilon_n = \max(|X_n|, |Y_n|, |Z_n|)$$

If $\epsilon_n < \text{tol}$, go to Stage 2; else compute

$$\lambda_n = (x_n y_n)^{1/2} + (x_n z_n)^{1/2} + (y_n z_n)^{1/2}$$
$$x_{n+1} = (x_n + \lambda_n)/4, \quad y_{n+1} = (y_n + \lambda_n)/4, \quad z_{n+1} = (z_n + \lambda_n)/4$$

and repeat this stage.

Stage 2: Compute

$$E_2 = X_n Y_n - Z_n^2, \quad E_3 = X_n Y_n Z_n$$
$$R_F = (1 - \tfrac{1}{10}E_2 + \tfrac{1}{14}E_3 + \tfrac{1}{24}E_2^2 - \tfrac{3}{44}E_2 E_3)/(\mu_n)^{1/2}$$

In some applications the argument p in R_J or the argument y in R_C is negative, and the Cauchy principal value of the integral is required. This is easily handled by using the formulas

$$R_J(x,y,z,p) = [(\gamma - y)R_J(x,y,z,\gamma) - 3R_F(x,y,z) + 3R_C(xz/y, p\gamma/y)]/(y-p) \tag{6.12.14}$$

where

$$\gamma \equiv y + \frac{(z-y)(y-x)}{y-p} \tag{6.12.15}$$

is positive if p is negative, and

$$R_C(x,y) = \left(\frac{x}{x-y}\right)^{1/2} R_C(x-y, -y) \tag{6.12.16}$$

The Cauchy principal value of R_J has a zero at some value of $p < 0$, so (6.12.14) will give some loss of significant figures near the zero.

elliptint.h

Doub rf(const Doub x, const Doub y, const Doub z) {

Computes Carlson's elliptic integral of the first kind, $R_F(x,y,z)$. x, y, and z must be nonnegative, and at most one can be zero.

```
    static const Doub ERRTOL=0.0025, THIRD=1.0/3.0,C1=1.0/24.0, C2=0.1,
        C3=3.0/44.0, C4=1.0/14.0;
    static const Doub TINY=5.0*numeric_limits<Doub>::min(),
        BIG=0.2*numeric_limits<Doub>::max();
    Doub alamb,ave,delx,dely,delz,e2,e3,sqrtx,sqrty,sqrtz,xt,yt,zt;
    if (MIN(MIN(x,y),z) < 0.0 || MIN(MIN(x+y,x+z),y+z) < TINY ||
        MAX(MAX(x,y),z) > BIG) throw("invalid arguments in rf");
    xt=x;
    yt=y;
    zt=z;
    do {
        sqrtx=sqrt(xt);
        sqrty=sqrt(yt);
        sqrtz=sqrt(zt);
        alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz;
        xt=0.25*(xt+alamb);
        yt=0.25*(yt+alamb);
        zt=0.25*(zt+alamb);
        ave=THIRD*(xt+yt+zt);
        delx=(ave-xt)/ave;
        dely=(ave-yt)/ave;
        delz=(ave-zt)/ave;
    } while (MAX(MAX(abs(delx),abs(dely)),abs(delz)) > ERRTOL);
    e2=delx*dely-delz*delz;
    e3=delx*dely*delz;
    return (1.0+(C1*e2-C2-C3*e3)*e2+C4*e3)/sqrt(ave);
}
```

A value of 0.0025 for the error tolerance parameter gives full double precision (16 significant digits). Since the error scales as ϵ_n^6, we see that 0.08 would be adequate for single precision (7 significant digits), but would save at most two or three more iterations. Since the coefficients of the sixth-order truncation error are different for the other elliptic functions, these values for the error tolerance should be set to 0.04 (single precision) or 0.0012 (double precision) in the algorithm for R_C and 0.05 or 0.0015 for R_J and R_D. As well as being an algorithm in its own right for certain combinations of elementary functions, the algorithm for R_C is used repeatedly in the computation of R_J.

The C++ implementations test the input arguments against two machine-dependent constants, `TINY` and `BIG`, to ensure that there will be no underflow or overflow during the computation. You can always extend the range of admissible argument values by using the homogeneity relations (6.12.22), below.

elliptint.h

Doub rd(const Doub x, const Doub y, const Doub z) {

Computes Carlson's elliptic integral of the second kind, $R_D(x,y,z)$. x and y must be nonnegative, and at most one can be zero. z must be positive.

```
    static const Doub ERRTOL=0.0015, C1=3.0/14.0, C2=1.0/6.0, C3=9.0/22.0,
        C4=3.0/26.0, C5=0.25*C3, C6=1.5*C4;
    static const Doub TINY=2.0*pow(numeric_limits<Doub>::max(),-2./3.),
        BIG=0.1*ERRTOL*pow(numeric_limits<Doub>::min(),-2./3.);
    Doub alamb,ave,delx,dely,delz,ea,eb,ec,ed,ee,fac,sqrtx,sqrty,
        sqrtz,sum,xt,yt,zt;
    if (MIN(x,y) < 0.0 || MIN(x+y,z) < TINY || MAX(MAX(x,y),z) > BIG)
        throw("invalid arguments in rd");
    xt=x;
    yt=y;
    zt=z;
    sum=0.0;
    fac=1.0;
    do {
        sqrtx=sqrt(xt);
```

```
        sqrty=sqrt(yt);
        sqrtz=sqrt(zt);
        alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz;
        sum += fac/(sqrtz*(zt+alamb));
        fac=0.25*fac;
        xt=0.25*(xt+alamb);
        yt=0.25*(yt+alamb);
        zt=0.25*(zt+alamb);
        ave=0.2*(xt+yt+3.0*zt);
        delx=(ave-xt)/ave;
        dely=(ave-yt)/ave;
        delz=(ave-zt)/ave;
    } while (MAX(MAX(abs(delx),abs(dely)),abs(delz)) > ERRTOL);
    ea=delx*dely;
    eb=delz*delz;
    ec=ea-eb;
    ed=ea-6.0*eb;
    ee=ed+ec+ec;
    return 3.0*sum+fac*(1.0+ed*(-C1+C5*ed-C6*delz*ee)
        +delz*(C2*ee+delz*(-C3*ec+delz*C4*ea)))/(ave*sqrt(ave));
}
```

```
Doub rj(const Doub x, const Doub y, const Doub z, const Doub p) {
```

Computes Carlson's elliptic integral of the third kind, $R_J(x, y, z, p)$. x, y, and z must be nonnegative, and at most one can be zero. p must be nonzero. If $p < 0$, the Cauchy principal value is returned.

elliptint.h

```
    static const Doub ERRTOL=0.0015, C1=3.0/14.0, C2=1.0/3.0, C3=3.0/22.0,
        C4=3.0/26.0, C5=0.75*C3, C6=1.5*C4, C7=0.5*C2, C8=C3+C3;
    static const Doub TINY=pow(5.0*numeric_limits<Doub>::min(),1./3.),
        BIG=0.3*pow(0.2*numeric_limits<Doub>::max(),1./3.);
    Doub a,alamb,alpha,ans,ave,b,beta,delp,delx,dely,delz,ea,eb,ec,ed,ee,
        fac,pt,rcx,rho,sqrtx,sqrty,sqrtz,sum,tau,xt,yt,zt;
    if (MIN(MIN(x,y),z) < 0.0 || MIN(MIN(x+y,x+z),MIN(y+z,abs(p))) < TINY
        || MAX(MAX(x,y),MAX(z,abs(p))) > BIG) throw("invalid arguments in rj");
    sum=0.0;
    fac=1.0;
    if (p > 0.0) {
        xt=x;
        yt=y;
        zt=z;
        pt=p;
    } else {
        xt=MIN(MIN(x,y),z);
        zt=MAX(MAX(x,y),z);
        yt=x+y+z-xt-zt;
        a=1.0/(yt-p);
        b=a*(zt-yt)*(yt-xt);
        pt=yt+b;
        rho=xt*zt/yt;
        tau=p*pt/yt;
        rcx=rc(rho,tau);
    }
    do {
        sqrtx=sqrt(xt);
        sqrty=sqrt(yt);
        sqrtz=sqrt(zt);
        alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz;
        alpha=SQR(pt*(sqrtx+sqrty+sqrtz)+sqrtx*sqrty*sqrtz);
        beta=pt*SQR(pt+alamb);
        sum += fac*rc(alpha,beta);
        fac=0.25*fac;
        xt=0.25*(xt+alamb);
        yt=0.25*(yt+alamb);
```

```
        zt=0.25*(zt+alamb);
        pt=0.25*(pt+alamb);
        ave=0.2*(xt+yt+zt+pt+pt);
        delx=(ave-xt)/ave;
        dely=(ave-yt)/ave;
        delz=(ave-zt)/ave;
        delp=(ave-pt)/ave;
    } while (MAX(MAX(abs(delx),abs(dely)),
        MAX(abs(delz),abs(delp))) > ERRTOL);
    ea=delx*(dely+delz)+dely*delz;
    eb=delx*dely*delz;
    ec=delp*delp;
    ed=ea-3.0*ec;
    ee=eb+2.0*delp*(ea-ec);
    ans=3.0*sum+fac*(1.0+ed*(-C1+C5*ed-C6*ee)+eb*(C7+delp*(-C8+delp*C4))
        +delp*ea*(C2-delp*C3)-C2*delp*ec)/(ave*sqrt(ave));
    if (p <= 0.0) ans=a*(b*ans+3.0*(rcx-rf(xt,yt,zt)));
    return ans;
}
```

elliptint.h

```
Doub rc(const Doub x, const Doub y) {
```

Computes Carlson's degenerate elliptic integral, $R_C(x, y)$. x must be nonnegative and y must be nonzero. If $y < 0$, the Cauchy principal value is returned.

```
    static const Doub ERRTOL=0.0012, THIRD=1.0/3.0, C1=0.3, C2=1.0/7.0,
        C3=0.375, C4=9.0/22.0;
    static const Doub TINY=5.0*numeric_limits<Doub>::min(),
        BIG=0.2*numeric_limits<Doub>::max(), COMP1=2.236/sqrt(TINY),
        COMP2=SQR(TINY*BIG)/25.0;
    Doub alamb,ave,s,w,xt,yt;
    if (x < 0.0 || y == 0.0 || (x+abs(y)) < TINY || (x+abs(y)) > BIG ||
        (y<-COMP1 && x > 0.0 && x < COMP2)) throw("invalid arguments in rc");
    if (y > 0.0) {
        xt=x;
        yt=y;
        w=1.0;
    } else {
        xt=x-y;
        yt= -y;
        w=sqrt(x)/sqrt(xt);
    }
    do {
        alamb=2.0*sqrt(xt)*sqrt(yt)+yt;
        xt=0.25*(xt+alamb);
        yt=0.25*(yt+alamb);
        ave=THIRD*(xt+yt+yt);
        s=(yt-ave)/ave;
    } while (abs(s) > ERRTOL);
    return w*(1.0+s*s*(C1+s*(C2+s*(C3+s*C4))))/sqrt(ave);
}
```

At times you may want to express your answer in Legendre's notation. Alternatively, you may be given results in that notation and need to compute their values with the programs given above. It is a simple matter to transform back and forth. The *Legendre elliptic integral of the first kind* is defined as

$$F(\phi, k) \equiv \int_0^{\phi} \frac{d\theta}{\sqrt{1 - k^2 \sin^2 \theta}} \tag{6.12.17}$$

The *complete elliptic integral of the first kind* is given by

$$K(k) \equiv F(\pi/2, k) \tag{6.12.18}$$

In terms of R_F and

$$F(\phi,k) = \sin\phi R_F(\cos^2\phi, 1-k^2\sin^2\phi, 1)$$
$$K(k) = R_F(0, 1-k^2, 1) \tag{6.12.19}$$

The *Legendre elliptic integral of the second kind* and the *complete elliptic integral of the second kind* are given by

$$\begin{aligned} E(\phi,k) &\equiv \int_0^\phi \sqrt{1-k^2\sin^2\theta}\, d\theta \\ &= \sin\phi R_F(\cos^2\phi, 1-k^2\sin^2\phi, 1) \\ &\quad -\tfrac{1}{3}k^2\sin^3\phi R_D(\cos^2\phi, 1-k^2\sin^2\phi, 1) \\ E(k) &\equiv E(\pi/2,k) = R_F(0,1-k^2,1) - \tfrac{1}{3}k^2 R_D(0,1-k^2,1) \end{aligned} \tag{6.12.20}$$

Finally, the *Legendre elliptic integral of the third kind* is

$$\begin{aligned} \Pi(\phi,n,k) &\equiv \int_0^\phi \frac{d\theta}{(1+n\sin^2\theta)\sqrt{1-k^2\sin^2\theta}} \\ &= \sin\phi R_F(\cos^2\phi, 1-k^2\sin^2\phi, 1) \\ &\quad -\tfrac{1}{3}n\sin^3\phi R_J(\cos^2\phi, 1-k^2\sin^2\phi, 1, 1+n\sin^2\phi) \end{aligned} \tag{6.12.21}$$

(Note that this sign convention for n is opposite that of Abramowitz and Stegun [13], and that their $\sin\alpha$ is our k.)

```
Doub ellf(const Doub phi, const Doub ak) {
```
Legendre elliptic integral of the first kind $F(\phi,k)$, evaluated using Carlson's function R_F. The argument ranges are $0 \le \phi \le \pi/2$, $0 \le k\sin\phi \le 1$.
```
    Doub s=sin(phi);
    return s*rf(SQR(cos(phi)),(1.0-s*ak)*(1.0+s*ak),1.0);
}
```
elliptint.h

```
Doub elle(const Doub phi, const Doub ak) {
```
Legendre elliptic integral of the second kind $E(\phi,k)$, evaluated using Carlson's functions R_D and R_F. The argument ranges are $0 \le \phi \le \pi/2$, $0 \le k\sin\phi \le 1$.
```
    Doub cc,q,s;
    s=sin(phi);
    cc=SQR(cos(phi));
    q=(1.0-s*ak)*(1.0+s*ak);
    return s*(rf(cc,q,1.0)-(SQR(s*ak))*rd(cc,q,1.0)/3.0);
}
```
elliptint.h

```
Doub ellpi(const Doub phi, const Doub en, const Doub ak) {
```
Legendre elliptic integral of the third kind $\Pi(\phi,n,k)$, evaluated using Carlson's functions R_J and R_F. (Note that the sign convention on n is opposite that of Abramowitz and Stegun.) The ranges of ϕ and k are $0 \le \phi \le \pi/2$, $0 \le k\sin\phi \le 1$.
```
    Doub cc,enss,q,s;
    s=sin(phi);
    enss=en*s*s;
    cc=SQR(cos(phi));
    q=(1.0-s*ak)*(1.0+s*ak);
    return s*(rf(cc,q,1.0)-enss*rj(cc,q,1.0,1.0+enss)/3.0);
}
```
elliptint.h

Carlson's functions are homogeneous of degree $-\frac{1}{2}$ and $-\frac{3}{2}$, so

$$R_F(\lambda x, \lambda y, \lambda z) = \lambda^{-1/2} R_F(x, y, z)$$
$$R_J(\lambda x, \lambda y, \lambda z, \lambda p) = \lambda^{-3/2} R_J(x, y, z, p) \tag{6.12.22}$$

Thus, to express a Carlson function in Legendre's notation, permute the first three arguments into ascending order, use homogeneity to scale the third argument to be 1, and then use equations (6.12.19) – (6.12.21).

6.12.1 Jacobian Elliptic Functions

The Jacobian elliptic function sn is defined as follows: Instead of considering the elliptic integral

$$u(y,k) \equiv u = F(\phi, k) \tag{6.12.23}$$

consider the *inverse* function

$$y = \sin\phi = \mathrm{sn}(u,k) \tag{6.12.24}$$

Equivalently,

$$u = \int_0^{\mathrm{sn}} \frac{dy}{\sqrt{(1-y^2)(1-k^2y^2)}} \tag{6.12.25}$$

When $k = 0$, sn is just sin. The functions cn and dn are defined by the relations

$$\mathrm{sn}^2 + \mathrm{cn}^2 = 1, \qquad k^2\mathrm{sn}^2 + \mathrm{dn}^2 = 1 \tag{6.12.26}$$

The routine given below actually takes $m_c \equiv k_c^2 = 1 - k^2$ as an input parameter. It also computes all three functions sn, cn, and dn since computing all three is no harder than computing any one of them. For a description of the method, see [9].

elliptint.h

```
void sncndn(const Doub uu, const Doub emmc, Doub &sn, Doub &cn, Doub &dn) {
```

Returns the Jacobian elliptic functions $\mathrm{sn}(u,k_c)$, $\mathrm{cn}(u,k_c)$, and $\mathrm{dn}(u,k_c)$. Here uu $= u$, while emmc $= k_c^2$.

```
    static const Doub CA=1.0e-8;
```

The accuracy is the square of CA.

```
    Bool bo;
    Int i,ii,l;
    Doub a,b,c,d,emc,u;
    VecDoub em(13),en(13);
    emc=emmc;
    u=uu;
    if (emc != 0.0) {
        bo=(emc < 0.0);
        if (bo) {
            d=1.0-emc;
            emc /= -1.0/d;
            u *= (d=sqrt(d));
        }
        a=1.0;
        dn=1.0;
        for (i=0;i<13;i++) {
            l=i;
            em[i]=a;
            en[i]=(emc=sqrt(emc));
            c=0.5*(a+emc);
            if (abs(a-emc) <= CA*a) break;
            emc *= a;
```

```
            a=c;
        }
        u *= c;
        sn=sin(u);
        cn=cos(u);
        if (sn != 0.0) {
            a=cn/sn;
            c *= a;
            for (ii=l;ii>=0;ii--) {
                b=em[ii];
                a *= c;
                c *= dn;
                dn=(en[ii]+a)/(b+a);
                a=c/b;
            }
            a=1.0/sqrt(c*c+1.0);
            sn=(sn >= 0.0 ? a : -a);
            cn=c*sn;
        }
        if (bo) {
            a=dn;
            dn=cn;
            cn=a;
            sn /= d;
        }
    } else {
        cn=1.0/cosh(u);
        dn=cn;
        sn=tanh(u);
    }
}
```

CITED REFERENCES AND FURTHER READING:

Erdélyi, A., Magnus, W., Oberhettinger, F., and Tricomi, F.G. 1953, *Higher Transcendental Functions*, Vol. II, (New York: McGraw-Hill).[1]

Gradshteyn, I.S., and Ryzhik, I.W. 1980, *Table of Integrals, Series, and Products* (New York: Academic Press).[2]

Carlson, B.C. 1977, "Elliptic Integrals of the First Kind," *SIAM Journal on Mathematical Analysis*, vol. 8, pp. 231–242.[3]

Carlson, B.C. 1987, "A Table of Elliptic Integrals of the Second Kind," *Mathematics of Computation*, vol. 49, pp. 595–606[4]; 1988, "A Table of Elliptic Integrals of the Third Kind," *op. cit.*, vol. 51, pp. 267–280[5]; 1989, "A Table of Elliptic Integrals: Cubic Cases," *op. cit.*, vol. 53, pp. 327–333[6]; 1991, "A Table of Elliptic Integrals: One Quadratic Factor," *op. cit.*, vol. 56, pp. 267–280.[7]

Carlson, B.C., and FitzSimons, J. 2000, "Reduction Theorems for Elliptic Integrands with the Square Root of Two Quadratic Factors," *Journal of Computational and Applied Mathematics*, vol. 118, pp. 71–85.[8]

Bulirsch, R. 1965, "Numerical Calculation of Elliptic Integrals and Elliptic Functions," *Numerische Mathematik*, vol. 7, pp. 78–90; 1965, *op. cit.*, vol. 7, pp. 353–354; 1969, *op. cit.*, vol. 13, pp. 305–315.[9]

Carlson, B.C. 1979, "Computing Elliptic Integrals by Duplication," *Numerische Mathematik*, vol. 33, pp. 1–16.[10]

Carlson, B.C., and Notis, E.M. 1981, "Algorithms for Incomplete Elliptic Integrals," *ACM Transactions on Mathematical Software*, vol. 7, pp. 398–403.[11]

Carlson, B.C. 1978, "Short Proofs of Three Theorems on Elliptic Integrals," *SIAM Journal on Mathematical Analysis*, vol. 9, p. 524–528.[12]

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at `http://www.nr.com/aands`, Chapter 17.[13]

Mathews, J., and Walker, R.L. 1970, *Mathematical Methods of Physics*, 2nd ed. (Reading, MA: W.A. Benjamin/Addison-Wesley), pp. 78–79.

6.13 Hypergeometric Functions

As was discussed in §5.14, a fast, general routine for the the complex hypergeometric function ${}_2F_1(a,b,c;z)$ is difficult or impossible. The function is defined as the analytic continuation of the hypergeometric series

$$ {}_2F_1(a,b,c;z) = 1 + \frac{ab}{c}\frac{z}{1!} + \frac{a(a+1)b(b+1)}{c(c+1)}\frac{z^2}{2!} + \cdots + \frac{a(a+1)\ldots(a+j-1)b(b+1)\ldots(b+j-1)}{c(c+1)\ldots(c+j-1)}\frac{z^j}{j!} + \cdots \tag{6.13.1} $$

This series converges only within the unit circle $|z| < 1$ (see [1]), but one's interest in the function is not confined to this region.

Section 5.14 discussed the method of evaluating this function by direct path integration in the complex plane. We here merely list the routines that result.

Implementation of the function `hypgeo` is straightforward and is described by comments in the program. The machinery associated with Chapter 17's routine for integrating differential equations, `Odeint`, is only minimally intrusive and need not even be completely understood: Use of `Odeint` requires one function call to the constructor, with a prescribed format for the derivative routine `Hypderiv`, followed by a call to the `integrate` method.

The function `hypgeo` will fail, of course, for values of z too close to the singularity at 1. (If you need to approach this singularity, or the one at ∞, use the "linear transformation formulas" in §15.3 of [1].) Away from $z = 1$ and for moderate values of a, b, c, it is often remarkable how few steps are required to integrate the equations. A half-dozen is typical.

hypgeo.h

```
Complex hypgeo(const Complex &a, const Complex &b,const Complex &c,
    const Complex &z)
```

Complex hypergeometric function ${}_2F_1$ for complex a, b, c, and z, by direct integration of the hypergeometric equation in the complex plane. The branch cut is taken to lie along the real axis, Re $z > 1$.

```
{
    const Doub atol=1.0e-14,rtol=1.0e-14;      Accuracy parameters.
    Complex ans,dz,z0,y[2];
    VecDoub yy(4);
    if (norm(z) <= 0.25) {                     Use series...
        hypser(a,b,c,z,ans,y[1]);
        return ans;
    }
    ...or pick a starting point for the path integration.
    else if (real(z) < 0.0) z0=Complex(-0.5,0.0);
    else if (real(z) <= 1.0) z0=Complex(0.5,0.0);
```

```
    else z0=Complex(0.0,imag(z) >= 0.0 ? 0.5 : -0.5);
    dz=z-z0;
    hypser(a,b,c,z0,y[0],y[1]);                    Get starting function and derivative.
    yy[0]=real(y[0]);
    yy[1]=imag(y[0]);
    yy[2]=real(y[1]);
    yy[3]=imag(y[1]);
    Hypderiv d(a,b,c,z0,dz);                       Set up the functor for the derivatives.
    Output out;                                    Suppress output in Odeint.
    Odeint<StepperBS<Hypderiv> > ode(yy,0.0,1.0,atol,rtol,0.1,0.0,out,d);
```

The arguments to `Odeint` are the vector of independent variables, the starting and ending values of the dependent variable, the accuracy parameters, an initial guess for the stepsize, a minimum stepsize, and the names of the output object and the derivative object. The integration is performed by the Bulirsch-Stoer stepping routine.

```
    ode.integrate();
    y[0]=Complex(yy[0],yy[1]);
    return y[0];
}
```

```
void hypser(const Complex &a, const Complex &b, const Complex &c,                    hypgeo.h
    const Complex &z, Complex &series, Complex &deriv)
```

Returns the hypergeometric series ${}_2F_1$ and its derivative, iterating to machine accuracy. For $|z| \leq 1/2$ convergence is quite rapid.

```
{
    deriv=0.0;
    Complex fac=1.0;
    Complex temp=fac;
    Complex aa=a;
    Complex bb=b;
    Complex cc=c;
    for (Int n=1;n<=1000;n++) {
        fac *= ((aa*bb)/cc);
        deriv += fac;
        fac *= ((1.0/n)*z);
        series=temp+fac;
        if (series == temp) return;
        temp=series;
        aa += 1.0;
        bb += 1.0;
        cc += 1.0;
    }
    throw("convergence failure in hypser");
}
```

```
struct Hypderiv {                                                                    hypgeo.h
```

Functor to compute derivatives for the hypergeometric equation; see text equation (5.14.4).

```
    Complex a,b,c,z0,dz;
    Hypderiv(const Complex &aa, const Complex &bb,
        const Complex &cc, const Complex &z00,
        const Complex &dzz) : a(aa),b(bb),c(cc),z0(z00),dz(dzz) {}
    void operator() (const Doub s, VecDoub_I &yy, VecDoub_O &dyyds) {
        Complex z,y[2],dyds[2];
        y[0]=Complex(yy[0],yy[1]);
        y[1]=Complex(yy[2],yy[3]);
        z=z0+s*dz;
        dyds[0]=y[1]*dz;
        dyds[1]=(a*b*y[0]-(c-(a+b+1.0)*z)*y[1])*dz/(z*(1.0-z));
        dyyds[0]=real(dyds[0]);
        dyyds[1]=imag(dyds[0]);
        dyyds[2]=real(dyds[1]);
        dyyds[3]=imag(dyds[1]);
    }
};
```

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at `http://www.nr.com/aands`.[1]

6.14 Statistical Functions

Certain special functions get frequent use because of their relation to common univariate statistical distributions, that is, probability densities in a single variable. In this section we survey a number of such common distributions in a unified way, giving, in each case, routines for computing the probability density function $p(x)$; the cumulative density function or *cdf*, written $P(< x)$; and the inverse of the cumulative density function $x(P)$. The latter function is needed for finding the values of x associated with specified *percentile points* or *quantiles* in significance tests, for example, the 0.5%, 5%, 95% or 99.5% points.

The emphasis of this section is on defining and computing these statistical functions. Section §7.3 is a related section that discusses how to generate random deviates from the distributions discussed here. We defer discussion of the actual use of these distributions in statistical tests to Chapter 14.

6.14.1 Normal (Gaussian) Distribution

If x is drawn from a *normal distribution* with mean μ and standard deviation σ, then we write

$$x \sim \mathrm{N}(\mu,\sigma), \qquad \sigma > 0$$
$$p(x) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{1}{2}\left[\frac{x-\mu}{\sigma}\right]^2\right) \tag{6.14.1}$$

with $p(x)$ the probability density function. Note the special use of the notation "$\sim$" in this section, which can be read as "is drawn from a distribution." The variance of the distribution is, of course, σ^2.

The cumulative distribution function is the probability of a value $\le x$. For the normal distribution, this is given in terms of the complementary error function by

$$\mathrm{cdf} \equiv P(< x) \equiv \int_{-\infty}^{x} p(x')dx' = \frac{1}{2}\mathrm{erfc}\left(-\frac{1}{\sqrt{2}}\left[\frac{x-\mu}{\sigma}\right]\right) \tag{6.14.2}$$

The inverse cdf can thus be calculated in terms of the inverse of erfc,

$$x(P) = \mu - \sqrt{2}\sigma\,\mathrm{erfc}^{-1}(2P) \tag{6.14.3}$$

The following structure implements the above relations.

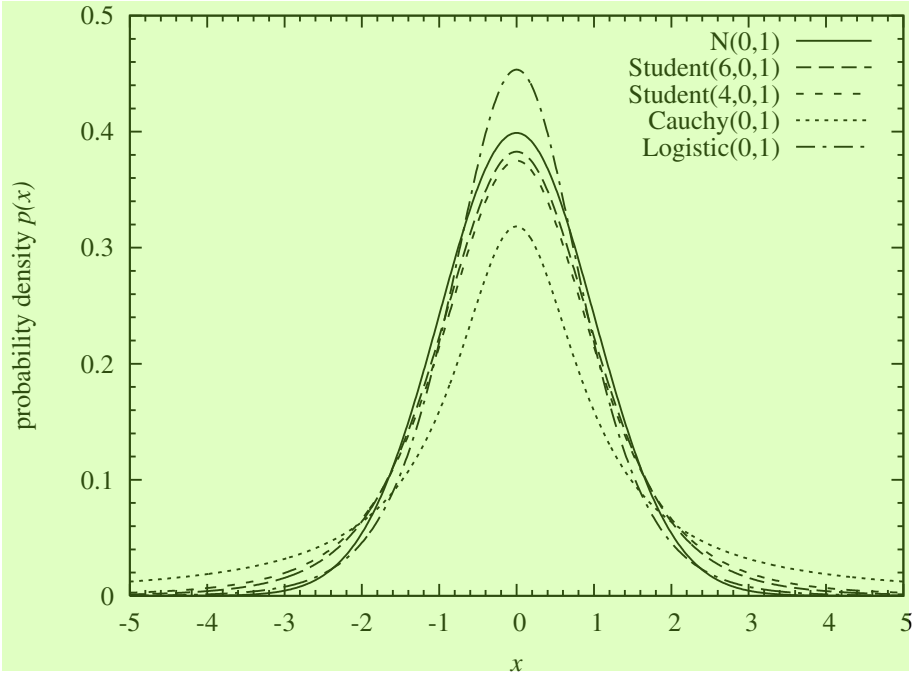

Figure 6.14.1. Examples of centrally peaked distributions that are symmetric on the real line. Any of these can substitute for the normal distribution either as an approximation or in applications such as robust estimation. They differ largely in the decay rate of their tails.

erf.h

```
struct Normaldist : Erf {
Normal distribution, derived from the error function Erf.
    Doub mu, sig;
    Normaldist(Doub mmu = 0., Doub ssig = 1.) : mu(mmu), sig(ssig) {
    Constructor. Initialize with μ and σ. The default with no arguments is N(0, 1).
        if (sig <= 0.) throw("bad sig in Normaldist");
    }
    Doub p(Doub x) {
    Return probability density function.
        return (0.398942280401432678/sig)*exp(-0.5*SQR((x-mu)/sig));
    }
    Doub cdf(Doub x) {
    Return cumulative distribution function.
        return 0.5*erfc(-0.707106781186547524*(x-mu)/sig);
    }
    Doub invcdf(Doub p) {
    Return inverse cumulative distribution function.
        if (p <= 0. || p >= 1.) throw("bad p in Normaldist");
        return -1.41421356237309505*sig*inverfc(2.*p)+mu;
    }
};
```

We will use the conventions of the above code for all the distributions in this section. A distribution's parameters (here, μ and σ) are set by the constructor and then referenced as needed by the member functions. The density function is always `p()`, the cdf is `cdf()`, and the inverse cdf is `invcdf()`. We will generally check the arguments of probability functions for validity, since many program bugs can show up as, e.g., a probability out of the range $[0, 1]$.

6.14.2 Cauchy Distribution

Like the normal distribution, the *Cauchy distribution* is a centrally peaked, symmetric distribution with a parameter μ that specifies its center and a parameter σ that specifies its width. *Unlike* the normal distribution, the Cauchy distribution has tails that decay very slowly at infinity, as $|x|^{-2}$, so slowly that moments higher than the zeroth moment (the area under the curve) don't even exist. The parameter μ is therefore, strictly speaking, not the mean, and the parameter σ is not, technically, the standard deviation. But these two parameters substitute for those moments as measures of central position and width.

The defining probability density is

$$x \sim \text{Cauchy}(\mu,\sigma), \qquad \sigma > 0$$
$$p(x) = \frac{1}{\pi\sigma}\left(1 + \left[\frac{x-\mu}{\sigma}\right]^2\right)^{-1} \tag{6.14.4}$$

If $x \sim \text{Cauchy}(0,1)$, then also $1/x \sim \text{Cauchy}(0,1)$ and also $(ax+b)^{-1} \sim \text{Cauchy}(-b/a, 1/a)$.

The cdf is given by

$$\text{cdf} \equiv P(<x) \equiv \int_{-\infty}^{x} p(x')dx' = \frac{1}{2} + \frac{1}{\pi}\arctan\left(\frac{x-\mu}{\sigma}\right) \tag{6.14.5}$$

The inverse cdf is given by

$$x(P) = \mu + \sigma\tan\left(\pi[P - \tfrac{1}{2}]\right) \tag{6.14.6}$$

Figure 6.14.1 shows Cauchy$(0,1)$ as compared to the normal distribution N$(0,1)$, as well as several other similarly shaped distributions discussed below.

The Cauchy distribution is sometimes called the *Lorentzian distribution*.

distributions.h

```
struct Cauchydist {
Cauchy distribution.
    Doub mu, sig;
    Cauchydist(Doub mmu = 0., Doub ssig = 1.) : mu(mmu), sig(ssig) {
    Constructor. Initialize with μ and σ. The default with no arguments is Cauchy(0, 1).
        if (sig <= 0.) throw("bad sig in Cauchydist");
    }
    Doub p(Doub x) {
    Return probability density function.
        return 0.318309886183790671/(sig*(1.+SQR((x-mu)/sig)));
    }
    Doub cdf(Doub x) {
    Return cumulative distribution function.
        return 0.5+0.318309886183790671*atan2(x-mu,sig);
    }
    Doub invcdf(Doub p) {
    Return inverse cumulative distribution function.
        if (p <= 0. || p >= 1.) throw("bad p in Cauchydist");
        return mu + sig*tan(3.14159265358979324*(p-0.5));
    }
};
```

6.14.3 Student-t Distribution

A generalization of the Cauchy distribution is the Student-t distribution, named for the early 20th century statistician William Gosset, who published under the name "Student" because his employer, Guinness Breweries, required him to use a pseudonym. Like the Cauchy distribution, the Student-t distribution has power-law tails at infinity, but it has an additional parameter ν that specifies how rapidly they decay, namely as $|t|^{-(\nu+1)}$. When ν is an integer, the number of convergent moments, including the zeroth, is thus ν.

The defining probability density (conventionally written in a variable t instead of x) is

$$t \sim \text{Student}(\nu,\mu,\sigma), \qquad \nu>0,\ \sigma>0$$

$$p(t) = \frac{\Gamma(\frac{1}{2}[\nu+1])}{\Gamma(\frac{1}{2}\nu)\sqrt{\nu\pi}\sigma}\left(1+\frac{1}{\nu}\left[\frac{t-\mu}{\sigma}\right]^2\right)^{-\frac{1}{2}(\nu+1)} \tag{6.14.7}$$

The Cauchy distribution is obtained in the case $\nu = 1$. In the opposite limit, $\nu \to \infty$, the normal distribution is obtained. In pre-computer days, this was the basis of various approximation schemes for the normal distribution, now all generally irrelevant. Figure 6.14.1 shows examples of the Student-t distribution for $\nu = 1$ (Cauchy), $\nu = 4$, and $\nu = 6$. The approach to the normal distribution is evident.

The mean of Student(ν,μ,σ) is (by symmetry) μ. The variance is not σ^2, but rather

$$\text{Var}\{\text{Student}(\nu,\mu,\sigma)\} = \frac{\nu}{\nu-2}\sigma^2 \tag{6.14.8}$$

For additional moments, and other properties, see [1].

The cdf is given by an incomplete beta function. If we let

$$x \equiv \frac{\nu}{\nu+\left(\frac{t-\mu}{\sigma}\right)^2} \tag{6.14.9}$$

then

$$\text{cdf} \equiv P(<t) \equiv \int_{-\infty}^{t} p(t')dt' = \begin{cases} \frac{1}{2}I_x(\frac{1}{2}\nu,\frac{1}{2}), & t \le \mu \\ 1-\frac{1}{2}I_x(\frac{1}{2}\nu,\frac{1}{2}), & t > \mu \end{cases} \tag{6.14.10}$$

The inverse cdf is given by an inverse incomplete beta function (see code below for the exact formulation).

In practice, the Student-t cdf is the above form is rarely used, since most statistical tests using Student-t are double-sided. Conventionally, the two-tailed function $A(t|\nu)$ is defined (only) for the case $\mu = 0$ and $\sigma = 1$ by

$$A(t|\nu) \equiv \int_{-t}^{+t} p(t')dt' = 1 - I_x(\tfrac{1}{2}\nu,\tfrac{1}{2}) \tag{6.14.11}$$

with x as given above. The statistic $A(t|\nu)$ is notably used in the test of whether two observed distributions have the same mean. The code below implements both equations (6.14.10) and (6.14.11), as well as their inverses.

ncgammabeta.h

```
struct Studenttdist : Beta {
```
Student-t distribution, derived from the beta function Beta.
```
    Doub nu, mu, sig, np, fac;
    Studenttdist(Doub nnu, Doub mmu = 0., Doub ssig = 1.)
    : nu(nnu), mu(mmu), sig(ssig) {
```
Constructor. Initialize with ν, μ and σ. The default with one argument is Student$(\nu, 0, 1)$.
```
        if (sig <= 0. || nu <= 0.) throw("bad sig,nu in Studentdist");
        np = 0.5*(nu + 1.);
        fac = gammln(np)-gammln(0.5*nu);
    }
    Doub p(Doub t) {
```
Return probability density function.
```
        return exp(-np*log(1.+SQR((t-mu)/sig)/nu)+fac)
            /(sqrt(3.14159265358979324*nu)*sig);
    }
    Doub cdf(Doub t) {
```
Return cumulative distribution function.
```
        Doub p = 0.5*betai(0.5*nu, 0.5, nu/(nu+SQR((t-mu)/sig)));
        if (t >= mu) return 1. - p;
        else return p;
    }
    Doub invcdf(Doub p) {
```
Return inverse cumulative distribution function.
```
        if (p <= 0. || p >= 1.) throw("bad p in Studentdist");
        Doub x = invbetai(2.*MIN(p,1.-p), 0.5*nu, 0.5);
        x = sig*sqrt(nu*(1.-x)/x);
        return (p >= 0.5? mu+x : mu-x);
    }
    Doub aa(Doub t) {
```
Return the two-tailed cdf $A(t|\nu)$.
```
        if (t < 0.) throw("bad t in Studentdist");
        return 1.-betai(0.5*nu, 0.5, nu/(nu+SQR(t)));
    }
    Doub invaa(Doub p) {
```
Return the inverse, namely t such that $p = A(t|\nu)$.
```
        if (p < 0. || p >= 1.) throw("bad p in Studentdist");
        Doub x = invbetai(1.-p, 0.5*nu, 0.5);
        return sqrt(nu*(1.-x)/x);
    }
};
```

6.14.4 Logistic Distribution 1

The *logistic distribution* is another symmetric, centrally peaked distribution that 3
can be used instead of the normal distribution. Its tails decay exponentially, but still much more slowly than the normal distribution's "exponent of the square."

The defining probability density is 4

$$p(y) = \frac{e^{-y}}{(1+e^{-y})^2} = \frac{e^{y}}{(1+e^{y})^2} = \tfrac{1}{4}\operatorname{sech}^2\left(\tfrac{1}{2}y\right) \quad (6.14.12)$$ 1 2

The three forms are algebraically equivalent, but, to avoid overflows, it is wise to 1
use the negative and positive exponential forms for positive and negative values of y, respectively.

The variance of the distribution (6.14.12) turns out to be $\pi^2/3$.2 Since it is 2
convenient to have parameters μ and σ with the conventional meanings of mean and standard deviation, equation (6.14.12) is often replaced by the *standardized logistic*

distribution,

$$x \sim \text{Logistic}(\mu, \sigma), \qquad \sigma > 0$$
$$p(x) = \frac{\pi}{4\sqrt{3}\sigma} \operatorname{sech}^2\left(\frac{\pi}{2\sqrt{3}}\left[\frac{x-\mu}{\sigma}\right]\right) \tag{6.14.13}$$

which implies equivalent forms using the positive and negative exponentials (see code below).

The cdf is given by

$$\text{cdf} \equiv P(< x) \equiv \int_{-\infty}^{x} p(x')dx' = \left[1 + \exp\left(-\frac{\pi}{\sqrt{3}}\left[\frac{x-\mu}{\sigma}\right]\right)\right]^{-1} \tag{6.14.14}$$

The inverse cdf is given by

$$x(P) = \mu + \frac{\sqrt{3}}{\pi}\sigma \log\left(\frac{P}{1-P}\right) \tag{6.14.15}$$

distributions.h

```
struct Logisticdist {
Logistic distribution.
    Doub mu, sig;
    Logisticdist(Doub mmu = 0., Doub ssig = 1.) : mu(mmu), sig(ssig) {
    Constructor. Initialize with μ and σ. The default with no arguments is Logistic(0, 1).
        if (sig <= 0.) throw("bad sig in Logisticdist");
    }
    Doub p(Doub x) {
    Return probability density function.
        Doub e = exp(-abs(1.81379936423421785*(x-mu)/sig));
        return 1.81379936423421785*e/(sig*SQR(1.+e));
    }
    Doub cdf(Doub x) {
    Return cumulative distribution function.
        Doub e = exp(-abs(1.81379936423421785*(x-mu)/sig));
        if (x >= mu) return 1./(1.+e);          Because we used abs to control over-
        else return e/(1.+e);                       flow, we now have two cases.
    }
    Doub invcdf(Doub p) {
    Return inverse cumulative distribution function.
        if (p <= 0. || p >= 1.) throw("bad p in Logisticdist");
        return mu + 0.5513288954217920 49*sig*log(p/(1.-p));
    }
};
```

The logistic distribution is cousin to the *logit transformation* that maps the open unit interval $0 < p < 1$ onto the real line $-\infty < u < \infty$ by the relation

$$u = \log\left(\frac{p}{1-p}\right) \tag{6.14.16}$$

Back when a book of tables and a slide rule were a statistician's working tools, the logit transformation was used to approximate processes on the interval by analytically simpler processes on the real line. A uniform distribution on the interval maps by the logit transformation to a logistic distribution on the real line. With the computer's ability to calculate distributions on the interval directly (beta distributions, for example), that motivation has vanished.

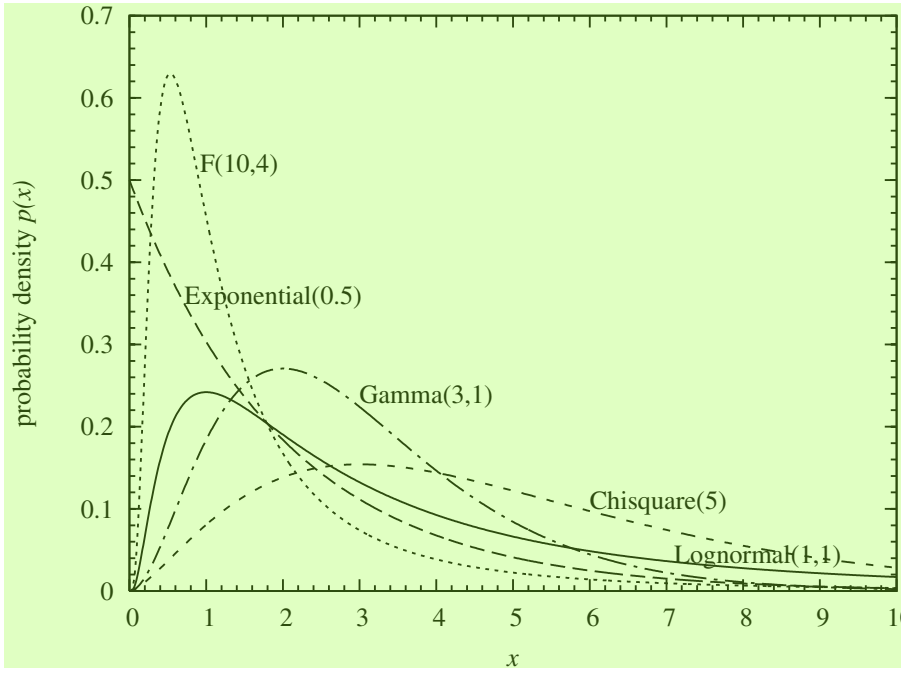

Figure 6.14.2. Examples of common distributions on the half-line $x > 0$.

Another cousin is the *logistic equation*,

$$\frac{dy}{dt} \propto y(y_{\max} - y) \tag{6.14.17}$$

a differential equation describing the growth of some quantity y, starting off as an exponential but reaching, asymptotically, a value $y_{\max}$. The solution of this equation is identical, up to a scaling, to the cdf of the logistic distribution.

6.14.5 Exponential Distribution

With the *exponential distribution* we now turn to common distribution functions defined on the positive real axis $x \geq 0$. Figure 6.14.2 shows examples of several of the distributions that we will discuss. The exponential is the simplest of them all. It has a parameter β that can control its width (in inverse relationship), but its mode is always at zero:

$$\begin{aligned} x &\sim \text{Exponential}(\beta), \qquad \beta > 0 \\ p(x) &= \beta \exp(-\beta x), \qquad x > 0 \end{aligned} \tag{6.14.18}$$

$$\text{cdf} \equiv P(< x) \equiv \int_0^x p(x')dx' = 1 - \exp(-\beta x) \tag{6.14.19}$$

$$x(P) = -\frac{1}{\beta}\log(1 - P) \tag{6.14.20}$$

The mean and standard deviation of the exponential distribution are both $1/\beta$. The median is $\log(2)/\beta$. Reference [1] has more to say about the exponential distribution than you would ever think possible.

distributions.h

```
struct Expondist {
Exponential distribution.
    Doub bet;
    Expondist(Doub bbet) : bet(bbet) {
    Constructor. Initialize with β.
        if (bet <= 0.) throw("bad bet in Expondist");
    }
    Doub p(Doub x) {
    Return probability density function.
        if (x < 0.) throw("bad x in Expondist");
        return bet*exp(-bet*x);
    }
    Doub cdf(Doub x) {
    Return cumulative distribution function.
        if (x < 0.) throw("bad x in Expondist");
        return 1.-exp(-bet*x);
    }
    Doub invcdf(Doub p) {
    Return inverse cumulative distribution function.
        if (p < 0. || p >= 1.) throw("bad p in Expondist");
        return -log(1.-p)/bet;
    }
};
```

6.14.6 Weibull Distribution

The Weibull distribution generalizes the exponential distribution in a way that is often useful in hazard, survival, or reliability studies. When the lifetime (time to failure) of an item is exponentially distributed, there is a constant probability per unit time that an item will fail, if it has not already done so. That is,

$$\text{hazard} \equiv \frac{p(x)}{P(>x)} \propto \text{constant} \tag{6.14.21}$$

Exponentially lived items don't age; they just keep rolling the same dice until, one day, their number comes up. In many other situations, however, it is observed that an item's hazard (as defined above) does change with time, say as a power law,

$$\frac{p(x)}{P(>x)} \propto x^{\alpha-1}, \qquad \alpha > 0 \tag{6.14.22}$$

The distribution that results is the Weibull distribution, named for Swedish physicist Waloddi Weibull, who used it as early as 1939. When $\alpha > 1$ the hazard increases with time, as for components that wear out. When $0 < \alpha < 1$ the hazard decreases with time, as for components that experience "infant mortality."

We say that

$$x \sim \text{Weibull}(\alpha, \beta) \qquad \text{iff} \qquad y \equiv \left(\frac{x}{\beta}\right)^{\alpha} \sim \text{Exponential}(1) \tag{6.14.23}$$

The probability density is

$$p(x) = \left(\frac{\alpha}{\beta}\right)\left(\frac{x}{\beta}\right)^{\alpha-1} e^{-(x/\beta)^{\alpha}}, \qquad x > 0 \tag{6.14.24}$$

The cdf is

$$\text{cdf} \equiv P(< x) \equiv \int_0^x p(x')dx' = 1 - e^{-(x/\beta)^\alpha} \tag{6.14.25}$$

The inverse cdf is

$$x(P) = \beta\,[-\log(1-P)]^{1/\alpha} \tag{6.14.26}$$

For $0 < \alpha < 1$, the distribution has an infinite (but integrable) cusp at $x = 0$ and is monotonically decreasing. The exponential distribution is the case of $\alpha = 1$. When $\alpha > 1$, the distribution is zero at $x = 0$ and has a single maximum at the value $x = \beta\,[(\alpha-1)/\alpha]^{1/\alpha}$.

The mean and variance are given by

$$\begin{aligned} \mu &= \beta\,\Gamma(1+\alpha^{-1}) \\ \sigma^2 &= \beta^2\left\{\Gamma(1+2\alpha^{-1}) - \left[\Gamma(1+\alpha^{-1})\right]^2\right\} \end{aligned} \tag{6.14.27}$$

With correct normalization, equation (6.14.22) becomes

$$\text{hazard} \equiv \frac{p(x)}{P(> x)} = \left(\frac{\alpha}{\beta}\right)\left(\frac{x}{\beta}\right)^{\alpha-1} \tag{6.14.28}$$

6.14.7 Lognormal Distribution

Many processes that live on the positive x-axis are naturally approximated by normal distributions on the "$\log(x)$-axis," that is, for $-\infty < \log(x) < \infty$. A simple, but important, example is the multiplicative random walk, which starts at some positive value x_0, and then generates new values by a recurrence like

$$x_{i+1} = \begin{cases} x_i(1+\epsilon) & \text{with probability } 0.5 \\ x_i/(1+\epsilon) & \text{with probability } 0.5 \end{cases} \tag{6.14.29}$$

Here ϵ is some small, fixed, constant.

These considerations motivate the definition

$$x \sim \text{Lognormal}(\mu,\sigma) \qquad \text{iff} \qquad u \equiv \frac{\log(x)-\mu}{\sigma} \sim \text{N}(0,1) \tag{6.14.30}$$

or the equivalent definition

$$x \sim \text{Lognormal}(\mu,\sigma), \qquad \sigma > 0$$

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma x}\exp\left(-\frac{1}{2}\left[\frac{\log(x)-\mu}{\sigma}\right]^2\right), \qquad x > 0 \tag{6.14.31}$$

Note the required extra factor of x^{-1} in front of the exponential: The density that is "normal" is $p(\log x)d\log x$.

While μ and σ are the mean and standard deviation in $\log x$ space, they are *not* so in x space. Rather,

$$\begin{aligned} \text{Mean}\{\text{Lognormal}(\mu,\sigma)\} &= e^{\mu+\frac{1}{2}\sigma^2} \\ \text{Var}\{\text{Lognormal}(\mu,\sigma)\} &= e^{2\mu}\,e^{\sigma^2}(e^{\sigma^2}-1) \end{aligned} \tag{6.14.32}$$

The cdf is given by

$$\text{cdf} \equiv P(< x) \equiv \int_0^x p(x')dx' = \frac{1}{2}\text{erfc}\left(-\frac{1}{\sqrt{2}}\left[\frac{\log(x)-\mu}{\sigma}\right]\right) \tag{6.14.33}$$

The inverse to the cdf involves the inverse complementary error function,

$$x(P) = \exp[\mu - \sqrt{2}\sigma\,\text{erfc}^{-1}(2P)] \tag{6.14.34}$$

erf.h

```
struct Lognormaldist : Erf {
Lognormal distribution, derived from the error function Erf.
    Doub mu, sig;
    Lognormaldist(Doub mmu = 0., Doub ssig = 1.) : mu(mmu), sig(ssig) {
        if (sig <= 0.) throw("bad sig in Lognormaldist");
    }
    Doub p(Doub x) {
    Return probability density function.
        if (x < 0.) throw("bad x in Lognormaldist");
        if (x == 0.) return 0.;
        return (0.398942280401432678/(sig*x))*exp(-0.5*SQR((log(x)-mu)/sig));
    }
    Doub cdf(Doub x) {
    Return cumulative distribution function.
        if (x < 0.) throw("bad x in Lognormaldist");
        if (x == 0.) return 0.;
        return 0.5*erfc(-0.707106781186547524*(log(x)-mu)/sig);
    }
    Doub invcdf(Doub p) {
    Return inverse cumulative distribution function.
        if (p <= 0. || p >= 1.) throw("bad p in Lognormaldist");
        return exp(-1.41421356237309505*sig*inverfc(2.*p)+mu);
    }
};
```

Multiplicative random walks like (6.14.29) and lognormal distributions are key ingredients in the economic theory of efficient markets, leading to (among many other results) the celebrated *Black-Scholes formula* for the probability distribution of the price of an investment after some elapsed time τ. A key piece of the Black-Scholes derivation is implicit in equation (6.14.32): If an investment's average return is zero (which may be true in the limit of zero risk), then its price cannot simply be a widening lognormal distribution with fixed μ and increasing σ, for its expected value would then diverge to infinity! The actual Black-Scholes formula thus defines both how σ increases with time (basically as $\tau^{1/2}$) *and* how μ correspondingly decreases with time, so as to keep the overall mean under control. A simplified version of the Black-Scholes formula can be written as

$$S(\tau) \sim S(0) \times \text{Lognormal}\left(r\tau - \tfrac{1}{2}\sigma^2\tau,\ \sigma\sqrt{\tau}\right) \tag{6.14.35}$$

where $S(\tau)$ is the price of a stock at time τ, r is its expected (annualized) rate of return, and σ is now redefined to be the stock's (annualized) volatility. The definition of volatility is that, for small values of τ, the fractional variance of the stock's price is $\sigma^2\tau$. You can check that (6.14.35) has the desired expectation value $E[S(\tau)] = S(0)$, for all τ, if $r = 0$. A good reference is [3].

6.14.8 Chi-Square Distribution

The *chi-square* (or χ^2) distribution has a single parameter $\nu > 0$ that controls both the location and width of its peak. In most applications ν is an integer and is referred to as the *number of degrees of freedom* (see §14.3).

The defining probability density is

$$\begin{aligned} \chi^2 &\sim \text{Chisquare}(\nu), \qquad \nu > 0 \\ p(\chi^2)d\chi^2 &= \frac{1}{2^{\frac{1}{2}\nu}\Gamma(\frac{1}{2}\nu)}(\chi^2)^{\frac{1}{2}\nu-1}\exp\left(-\tfrac{1}{2}\chi^2\right)d\chi^2, \qquad \chi^2 > 0 \end{aligned} \tag{6.14.36}$$

where we have written the differentials $d\chi^2$ merely to emphasize that χ^2, not χ, is to be viewed as the independent variable.

The mean and variance are given by

$$\begin{aligned} \text{Mean}\{\text{Chisquare}(\nu)\} &= \nu \\ \text{Var}\{\text{Chisquare}(\nu)\} &= 2\nu \end{aligned} \tag{6.14.37}$$

When $\nu \geq 2$ there is a single mode at $\chi^2 = \nu - 2$.

The chi-square distribution is actually just a special case of the gamma distribution, below, so its cdf is given by an incomplete gamma function $P(a,x)$:

$$\text{cdf} \equiv P(<\chi^2) \equiv P(\chi^2|\nu) \equiv \int_0^{\chi^2} p(\chi^{2\prime})d\chi^{2\prime} = P\left(\frac{\nu}{2},\frac{\chi^2}{2}\right) \tag{6.14.38}$$

One frequently also sees the complement of the cdf, which can be calculated either from the incomplete gamma function $P(a,x)$, or from its complement $Q(a,x)$ (often more accurate if P is very close to 1):

$$Q(\chi^2|\nu) \equiv 1 - P(\chi^2|n) = 1 - P\left(\frac{\nu}{2},\frac{\chi^2}{2}\right) \equiv Q\left(\frac{\nu}{2},\frac{\chi^2}{2}\right) \tag{6.14.39}$$

The inverse cdf is given in terms of the function that is the inverse of $P(a,x)$ on its second argument, which we here denote $P^{-1}(a,p)$:

$$x(P) = 2P^{-1}\left(\frac{\nu}{2},P\right) \tag{6.14.40}$$

ncgammabeta.h

```
struct Chisqdist : Gamma {
χ² distribution, derived from the gamma function Gamma.
    Doub nu,fac;
    Chisqdist(Doub nnu) : nu(nnu) {
    Constructor. Initialize with ν.
        if (nu <= 0.) throw("bad nu in Chisqdist");
        fac = 0.693147180559945309*(0.5*nu)+gammln(0.5*nu);
    }
    Doub p(Doub x2) {
    Return probability density function.
        if (x2 <= 0.) throw("bad x2 in Chisqdist");
        return exp(-0.5*(x2-(nu-2.)*log(x2))-fac);
    }
    Doub cdf(Doub x2) {
```

```
    Return cumulative distribution function.
        if (x2 < 0.) throw("bad x2 in Chisqdist");
        return gammp(0.5*nu,0.5*x2);
    }
    Doub invcdf(Doub p) {
    Return inverse cumulative distribution function.
        if (p < 0. || p >= 1.) throw("bad p in Chisqdist");
        return 2.*invgammp(p,0.5*nu);
    }
};
```

6.14.9 Gamma Distribution

The *gamma distribution* is defined by

$$x \sim \text{Gamma}(\alpha,\beta), \qquad \alpha>0,\ \beta>0$$
$$p(x) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x}, \qquad x>0 \tag{6.14.41}$$

The exponential distribution is the special case with $\alpha = 1$. The chi-square distribution is the special case with $\alpha = \nu/2$ and $\beta = 1/2$.

The mean and variance are given by,

$$\text{Mean}\{\text{Gamma}(\alpha,\beta)\} = \alpha/\beta$$
$$\text{Var}\{\text{Gamma}(\alpha,\beta)\} = \alpha/\beta^2 \tag{6.14.42}$$

When $\alpha \geq 1$ there is a single mode at $x = (\alpha-1)/\beta$.

Evidently, the cdf is the incomplete gamma function

$$\text{cdf} \equiv P(<x) \equiv \int_0^x p(x')dx' = P(\alpha,\beta x) \tag{6.14.43}$$

while the inverse cdf is given in terms of the inverse of $P(a,x)$ on its second argument by

$$x(P) = \frac{1}{\beta} P^{-1}(\alpha, P) \tag{6.14.44}$$

incgammabeta.h

```
struct Gammadist : Gamma {
Gamma distribution, derived from the gamma function Gamma.
    Doub alph, bet, fac;
    Gammadist(Doub aalph, Doub bbet = 1.) : alph(aalph), bet(bbet) {
    Constructor. Initialize with α and β.
        if (alph <= 0. || bet <= 0.) throw("bad alph,bet in Gammadist");
        fac = alph*log(bet)-gammln(alph);
    }
    Doub p(Doub x) {
    Return probability density function.
        if (x <= 0.) throw("bad x in Gammadist");
        return exp(-bet*x+(alph-1.)*log(x)+fac);
    }
    Doub cdf(Doub x) {
    Return cumulative distribution function.
        if (x < 0.) throw("bad x in Gammadist");
        return gammp(alph,bet*x);
    }
```

```
    Doub invcdf(Doub p) {
    Return inverse cumulative distribution function.
        if (p < 0. || p >= 1.) throw("bad p in Gammadist");
        return invgammp(p,alph)/bet;
    }
};
```

6.14.10 F-Distribution

The *F-distribution* is parameterized by two positive values ν_1 and ν_2, usually (but not always) integers.

The defining probability density is

$$\begin{aligned} F &\sim \mathrm{F}(\nu_1,\nu_2), \qquad \nu_1>0,\ \nu_2>0 \\ p(F) &= \frac{\nu_1^{\frac{1}{2}\nu_1}\,\nu_2^{\frac{1}{2}\nu_2}}{B(\frac{1}{2}\nu_1,\frac{1}{2}\nu_2)}\,\frac{F^{\frac{1}{2}\nu_1-1}}{(\nu_2+\nu_1 F)^{(\nu_1+\nu_2)/2}}, \qquad F>0 \end{aligned} \tag{6.14.45}$$

where $B(a,b)$ denotes the beta function. The mean and variance are given by

$$\begin{aligned} \mathrm{Mean}\{\mathrm{F}(\nu_1,\nu_2)\} &= \frac{\nu_2}{\nu_2-2}, \qquad \nu_2>2 \\ \mathrm{Var}\{\mathrm{F}(\nu_1,\nu_2)\} &= \frac{2\nu_2^2(\nu_1+\nu_2-2)}{\nu_1(\nu_2-2)^2(\nu_2-4)}, \qquad \nu_2>4 \end{aligned} \tag{6.14.46}$$

When $\nu_1 \geq 2$, there is a single mode at

$$F = \frac{\nu_2(\nu_1-2)}{\nu_1(\nu_2+2)} \tag{6.14.47}$$

.

For fixed ν_1, if $\nu_2 \to \infty$, the F-distribution becomes a chi-square distribution, namely

$$\lim_{\nu_2\to\infty} \mathrm{F}(\nu_1,\nu_2) \cong \frac{1}{\nu_1}\mathrm{Chisquare}(\nu_1) \tag{6.14.48}$$

where "$\cong$" means "are identical distributions."

The F-distribution's cdf is given in terms of the incomplete beta function $I_x(a,b)$ by

$$\mathrm{cdf} \equiv P(<x) \equiv \int_0^x p(x')dx' = I_{\nu_1 F/(\nu_2+\nu_1 F)}\left(\tfrac{1}{2}\nu_1,\tfrac{1}{2}\nu_2\right) \tag{6.14.49}$$

while the inverse cdf is given in terms of the inverse of $I_x(a,b)$ on its subscript argument by

$$\begin{aligned} u &\equiv I_p^{-1}\left(\tfrac{1}{2}\nu_1,\tfrac{1}{2}\nu_2\right) \\ x(P) &= \frac{\nu_2 u}{\nu_1(1-u)} \end{aligned} \tag{6.14.50}$$

A frequent use of the F-distribution is to test whether two observed samples have the same variance.

incgammabeta.h

```
struct Fdist : Beta {
F distribution, derived from the beta function Beta.
    Doub nu1,nu2;
    Doub fac;
    Fdist(Doub nnu1, Doub nnu2) : nu1(nnu1), nu2(nnu2) {
    Constructor. Initialize with ν1 and ν2.
        if (nu1 <= 0. || nu2 <= 0.) throw("bad nu1,nu2 in Fdist");
        fac = 0.5*(nu1*log(nu1)+nu2*log(nu2))+gammln(0.5*(nu1+nu2))
            -gammln(0.5*nu1)-gammln(0.5*nu2);
    }
    Doub p(Doub f) {
    Return probability density function.
        if (f <= 0.) throw("bad f in Fdist");
        return exp((0.5*nu1-1.)*log(f)-0.5*(nu1+nu2)*log(nu2+nu1*f)+fac);
    }
    Doub cdf(Doub f) {
    Return cumulative distribution function.
        if (f < 0.) throw("bad f in Fdist");
        return betai(0.5*nu1,0.5*nu2,nu1*f/(nu2+nu1*f));
    }
    Doub invcdf(Doub p) {
    Return inverse cumulative distribution function.
        if (p <= 0. || p >= 1.) throw("bad p in Fdist");
        Doub x = invbetai(p,0.5*nu1,0.5*nu2);
        return nu2*x/(nu1*(1.-x));
    }
};
```

6.14.11 Beta Distribution

The *beta distribution* is defined on the unit interval $0 < x < 1$ by

$$\begin{aligned} x &\sim \text{Beta}(\alpha,\beta), \qquad \alpha > 0,\ \beta > 0 \\ p(x) &= \frac{1}{B(\alpha,\beta)} x^{\alpha-1}(1-x)^{\beta-1}, \qquad 0 < x < 1 \end{aligned} \tag{6.14.51}$$

The mean and variance are given by

$$\begin{aligned} \text{Mean}\{\text{Beta}(\alpha,\beta)\} &= \frac{\alpha}{\alpha+\beta} \\ \text{Var}\{\text{Beta}(\alpha,\beta)\} &= \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)} \end{aligned} \tag{6.14.52}$$

When $\alpha > 1$ and $\beta > 1$, there is a single mode at $(\alpha-1)/(\alpha+\beta-2)$. When $\alpha < 1$ and $\beta < 1$, the distribution function is "U-shaped" with a minimum at this same value. In other cases there is neither a maximum nor a minimum.

In the limit that β becomes large as α is held fixed, all the action in the beta distribution shifts toward $x = 0$, and the density function takes the shape of a gamma distribution. More precisely,

$$\lim_{\beta\to\infty} \beta\,\text{Beta}(\alpha,\beta) \cong \text{Gamma}(\alpha,1) \tag{6.14.53}$$

The cdf is the incomplete beta function

$$\text{cdf} \equiv P(<x) \equiv \int_0^x p(x')dx' = I_x(\alpha,\beta) \tag{6.14.54}$$

while the inverse cdf is given in terms of the inverse of $I_x(\alpha,\beta)$ on its subscript argument by

$$x(P) = I_p^{-1}(\alpha,\beta) \tag{6.14.55}$$

ncgammabeta.h

```
struct Betadist : Beta {
Beta distribution, derived from the beta function Beta.
    Doub alph, bet, fac;
    Betadist(Doub aalph, Doub bbet) : alph(aalph), bet(bbet) {
    Constructor. Initialize with α and β.
        if (alph <= 0. || bet <= 0.) throw("bad alph,bet in Betadist");
        fac = gammln(alph+bet)-gammln(alph)-gammln(bet);
    }
    Doub p(Doub x) {
    Return probability density function.
        if (x <= 0. || x >= 1.) throw("bad x in Betadist");
        return exp((alph-1.)*log(x)+(bet-1.)*log(1.-x)+fac);
    }
    Doub cdf(Doub x) {
    Return cumulative distribution function.
        if (x < 0. || x > 1.) throw("bad x in Betadist");
        return betai(alph,bet,x);
    }
    Doub invcdf(Doub p) {
    Return inverse cumulative distribution function.
        if (p < 0. || p > 1.) throw("bad p in Betadist");
        return invbetai(p,alph,bet);
    }
};
```

6.14.12 Kolmogorov-Smirnov Distribution

The *Kolmogorov-Smirnov* or *KS* distribution, defined for positive z, is key to an important statistical test that is discussed in §14.3. Its probability density function does not directly enter into the test and is virtually never even written down. What one typically needs to compute is the cdf, denoted $P_{KS}(z)$, or its complement, $Q_{KS}(z) \equiv 1 - P_{KS}(z)$.

The cdf $P_{KS}(z)$ is defined by the series

$$P_{KS}(z) = 1 - 2\sum_{j=1}^{\infty}(-1)^{j-1}\exp(-2j^2z^2) \tag{6.14.56}$$

or by the equivalent series (nonobviously so!)

$$P_{KS}(z) = \frac{\sqrt{2\pi}}{z}\sum_{j=1}^{\infty}\exp\left(-\frac{(2j-1)^2\pi^2}{8z^2}\right) \tag{6.14.57}$$

Limiting values are what you'd expect for cdf's named "P" and "Q":

$$\begin{aligned} P_{KS}(0) &= 0 & P_{KS}(\infty) &= 1 \\ Q_{KS}(0) &= 1 & Q_{KS}(\infty) &= 0 \end{aligned} \tag{6.14.58}$$

Both of the series (6.14.56) and (6.14.57) are convergent for all $z > 0$. Moreover, for any z, one or the other series converges extremely rapidly, requiring no more

than three terms to get to IEEE double precision fractional accuracy. A good place to switch from one series to the other is at $z \approx 1.18$. This renders the KS functions computable by a single exponential and a small number of arithmetic operations (see code below).

Getting the inverse functions $P_{KS}^{-1}(P)$ and $Q_{KS}^{-1}(Q)$, which return a value of z from a P or Q value, is a little trickier. For $Q \lesssim 0.3$ (that is, $P \gtrsim 0.7$), an iteration based on (6.14.56) works nicely:

$$\begin{aligned} x_0 &\equiv 0 \\ x_{i+1} &= \tfrac{1}{2}Q + x_i^4 - x_i^9 + x_i^{16} - x_i^{25} + \cdots \\ z(Q) &= \sqrt{-\tfrac{1}{2}\log(x_\infty)} \end{aligned} \qquad (6.14.59)$$

For $x \lesssim 0.06$ you only need the first two powers of x_i.

For larger values of Q, that is, $P \lesssim 0.7$, the number of powers of x required quickly becomes excessive. A useful approach is to write (6.14.57) as

$$\begin{aligned} y\log(y) &= -\frac{\pi P^2}{8}\left(1 + y^4 + y^{12} + \cdots + y^{2j(j-1)} + \cdots\right)^{-1} \\ z(P) &= \frac{\pi/2}{\sqrt{-\log(y)}} \end{aligned} \qquad (6.14.60)$$

If we can get a good enough initial guess for y, we can solve the first equation in (6.14.60) by a variant of Halley's method: Use values of y from the *previous* iteration on the right-hand side of (6.14.60), and use Halley only for the $y\log(y)$ piece, so that the first and second derivatives are analytically simple functions.

A good initial guess is obtained by using the inverse function to $y\log(y)$ (the function `invxlogx` in §6.11) with the argument $-\pi P^2/8$. The number of iterations within the `invxlogx` function and the Halley loop is never more than half a dozen in each, often less. Code for the KS functions and their inverses follows.

ksdist.h

```
struct KSdist {
Kolmogorov-Smirnov cumulative distribution functions and their inverses.
    Doub pks(Doub z) {
    Return cumulative distribution function.
        if (z < 0.) throw("bad z in KSdist");
        if (z == 0.) return 0.;
        if (z < 1.18) {
            Doub y = exp(-1.2337005501361698/SQR(z));
            return 2.25675833419102515*sqrt(-log(y))
                *(y + pow(y,9) + pow(y,25) + pow(y,49));
        } else {
            Doub x = exp(-2.*SQR(z));
            return 1. - 2.*(x - pow(x,4) + pow(x,9));
        }
    }
    Doub qks(Doub z) {
    Return complementary cumulative distribution function.
        if (z < 0.) throw("bad z in KSdist");
        if (z == 0.) return 1.;
        if (z < 1.18) return 1.-pks(z);
        Doub x = exp(-2.*SQR(z));
        return 2.*(x - pow(x,4) + pow(x,9));
    }
    Doub invqks(Doub q) {
```

```
    Return inverse of the complementary cumulative distribution function.
        Doub y,logy,yp,x,xp,f,ff,u,t;
        if (q <= 0. || q > 1.) throw("bad q in KSdist");
        if (q == 1.) return 0.;
        if (q > 0.3) {
            f = -0.39269908169872415*SQR(1.-q);
            y = invxlogx(f);                    Initial guess.
            do {
                yp = y;
                logy = log(y);
                ff = f/SQR(1.+ pow(y,4)+ pow(y,12));
                u = (y*logy-ff)/(1.+logy);      Newton's method correction.
                y = y - (t=u/MAX(0.5,1.-0.5*u/(y*(1.+logy))));  Halley.
            } while (abs(t/y)>1.e-15);
            return 1.57079632679489662/sqrt(-log(y));
        } else {
            x = 0.03;
            do {                                Iteration (6.14.59).
                xp = x;
                x = 0.5*q+pow(x,4)-pow(x,9);
                if (x > 0.06) x += pow(x,16)-pow(x,25);
            } while (abs((xp-x)/x)>1.e-15);
            return sqrt(-0.5*log(x));
        }
    }
    Doub invpks(Doub p) {return invqks(1.-p);}
    Return inverse of the cumulative distribution function.
};
```

6.14.13 Poisson Distribution

The eponymous *Poisson distribution* was derived by Poisson in 1837. It applies to a process where discrete, uncorrelated events occur at some mean rate per unit time. If, for a given period, λ is the mean expected number of events, then the probability distribution of seeing exactly k events, $k \geq 0$, can be written as

$$\begin{aligned} k &\sim \text{Poisson}(\lambda), \qquad \lambda > 0 \\ p(k) &= \frac{1}{k!}\lambda^k e^{-\lambda}, \qquad k = 0, 1, \ldots \end{aligned} \tag{6.14.61}$$

Evidently $\sum_k p(k) = 1$, since the k-dependent factors in (6.14.61) are just the series expansion of e^{λ}.

The mean and variance of Poisson(λ) are both λ. There is a single mode at $k = \lfloor \lambda \rfloor$, that is, at λ rounded down to an integer.

The Poisson distribution's cdf is an incomplete gamma function $Q(a, x)$,

$$P_\lambda(< k) = Q(k, \lambda) \tag{6.14.62}$$

Since k is discrete, $P_\lambda(< k)$ is of course different from $P_\lambda(\leq k)$, the latter being given by

$$P_\lambda(\leq k) = Q(k + 1, \lambda) \tag{6.14.63}$$

Some particular values are

$$P_\lambda(< 0) = 0 \qquad P_\lambda(< 1) = e^{-\lambda} \qquad P_\lambda(< \infty) = 1 \tag{6.14.64}$$

Some other relations involving the incomplete gamma functions $Q(a,x)$ and $P(a,x)$ are

$$P_\lambda(\geq k) = P(k,\lambda) = 1 - Q(k,\lambda)$$
$$P_\lambda(> k) = P(k+1,\lambda) = 1 - Q(k+1,\lambda) \tag{6.14.65}$$

Because of the discreteness in k, the inverse of the cdf must be defined with some care: Given a value P, we define $k_\lambda(P)$ as the integer such that

$$P_\lambda(< k) \leq P < P_\lambda(\leq k) \tag{6.14.66}$$

In the interest of conciseness, the code below cheats a little bit and allows the right-hand $<$ to be $\leq$. If you may be supplying P's that are *exact* $P_\lambda(< k)$'s, then you will need to check both $k_\lambda(P)$ as returned, and $k_\lambda(P)+1$. (This will essentially never happen for "round" P's like 0.95, 0.99, etc.)

incgammabeta.h

```
struct Poissondist : Gamma {
Poisson distribution, derived from the gamma function Gamma.
    Doub lam;
    Poissondist(Doub llam) : lam(llam) {
    Constructor. Initialize with λ.
        if (lam <= 0.) throw("bad lam in Poissondist");
    }
    Doub p(Int n) {
    Return probability density function.
        if (n < 0) throw("bad n in Poissondist");
        return exp(-lam + n*log(lam) - gammln(n+1.));
    }
    Doub cdf(Int n) {
    Return cumulative distribution function.
        if (n < 0) throw("bad n in Poissondist");
        if (n == 0) return 0.;
        return gammq((Doub)n,lam);
    }
    Int invcdf(Doub p) {
    Given argument P, return integer n such that P(< n) ≤ P ≤ P(< n + 1).
        Int n,nl,nu,inc=1;
        if (p <= 0. || p >= 1.) throw("bad p in Poissondist");
        if (p < exp(-lam)) return 0;
        n = (Int)MAX(sqrt(lam),5.);                Starting guess near peak of density.
        if (p < cdf(n)) {                          Expand interval until we bracket.
            do {
                n = MAX(n-inc,0);
                inc *= 2;
            } while (p < cdf(n));
            nl = n; nu = n + inc/2;
        } else {
            do {
                n += inc;
                inc *= 2;
            } while (p > cdf(n));
            nu = n; nl = n - inc/2;
        }
        while (nu-nl>1) {                          Now contract the interval by bisection.
            n = (nl+nu)/2;
            if (p < cdf(n)) nu = n;
            else nl = n;
        }
        return nl;
    }
};
```

6.14.14 Binomial Distribution

Like the Poisson distribution, the *binomial distribution* is a discrete distribution over $k \geq 0$. It has two parameters, $n \geq 1$, the "sample size" or maximum value for which k can be nonzero; and p, the "event probability" (not to be confused with $p(k)$, the probability of a particular k). We write

$$k \sim \text{Binomial}(n, p), \qquad n \geq 1,\ 0 < p < 1$$
$$p(k) = \binom{n}{k} p^k (1-p)^{n-k}, \qquad k = 0, 1, \ldots, n \tag{6.14.67}$$

where $\binom{n}{k}$ is, of course, the binomial coefficient.

The mean and variance are given by

$$\text{Mean}\{\text{Binomial}(n, p)\} = np$$
$$\text{Var}\{\text{Binomial}(n, p)\} = np(1-p) \tag{6.14.68}$$

There is a single mode at the value k that satisfies

$$(n+1)p - 1 < k \leq (n+1)p \tag{6.14.69}$$

The distribution is symmetrical iff $p = \frac{1}{2}$. Otherwise it has positive skewness for $p < \frac{1}{2}$ and negative for $p > \frac{1}{2}$. Many additional properties are described in [2].

The Poisson distribution is obtained from the binomial distribution in the limit $n \to \infty,\ p \to 0$ with the np remaining finite. More precisely,

$$\lim_{n\to\infty} \text{Binomial}(n, \lambda/n) \cong \text{Poisson}(\lambda) \tag{6.14.70}$$

The binomial distribution's cdf can be computed from the incomplete beta function $I_x(a, b)$:

$$P(< k) = 1 - I_p(k, n-k+1) \tag{6.14.71}$$

so we also have (analogously to the Poisson distribution)

$$P(\leq k) = 1 - I_p(k+1, n-k)$$
$$P(> k) = I_p(k+1, n-k)$$
$$P(\geq k) = I_p(k, n-k+1) \tag{6.14.72}$$

Some particular values are

$$P(< 0) = 0 \qquad P(< [n+1]) = 1 \tag{6.14.73}$$

The inverse cdf is defined exactly as for the Poisson distribution, above, and with the same small warning about the code.

ncgammabeta.h

```
struct Binomialdist : Beta {
Binomial distribution, derived from the beta function Beta.
    Int n;
    Doub pe, fac;
    Binomialdist(Int nn, Doub ppe) : n(nn), pe(ppe) {
    Constructor. Initialize with n (sample size) and p (event probability).
        if (n <= 0 || pe <= 0. || pe >= 1.) throw("bad args in Binomialdist");
```

```
        fac = gammln(n+1.);
    }
    Doub p(Int k) {
    Return probability density function.
        if (k < 0) throw("bad k in Binomialdist");
        if (k > n) return 0.;
        return exp(k*log(pe)+(n-k)*log(1.-pe)
            +fac-gammln(k+1.)-gammln(n-k+1.));
    }
    Doub cdf(Int k) {
    Return cumulative distribution function.
        if (k < 0) throw("bad k in Binomialdist");
        if (k == 0) return 0.;
        if (k > n) return 1.;
        return 1. - betai((Doub)k,n-k+1.,pe);
    }
    Int invcdf(Doub p) {
    Given argument P, return integer n such that P(< n) ≤ P ≤ P(< n + 1).
        Int k,kl,ku,inc=1;
        if (p <= 0. || p >= 1.) throw("bad p in Binomialdist");
        k = MAX(0,MIN(n,(Int)(n*pe)));          Starting guess near peak of density.
        if (p < cdf(k)) {                       Expand interval until we bracket.
            do {
                k = MAX(k-inc,0);
                inc *= 2;
            } while (p < cdf(k));
            kl = k; ku = k + inc/2;
        } else {
            do {
                k = MIN(k+inc,n+1);
                inc *= 2;
            } while (p > cdf(k));
            ku = k; kl = k - inc/2;
        }
        while (ku-kl>1) {                       Now contract the interval by bisection.
            k = (kl+ku)/2;
            if (p < cdf(k)) ku = k;
            else kl = k;
        }
        return kl;
    }
};
```

CITED REFERENCES AND FURTHER READING:

Johnson, N.L. and Kotz, S. 1970, *Continuous Univariate Distributions*, 2 vols. (Boston: Houghton Mifflin).[1]

Johnson, N.L. and Kotz, S. 1969, *Discrete Distributions* (Boston: Houghton Mifflin).[2]

Gelman, A., Carlin, J.B., Stern, H.S., and Rubin, D.B. 2003, *Bayesian Data Analysis*, 2nd ed. (Boca Raton, FL: Chapman & Hall/CRC), Appendix A.

Lyuu, Y-D. 2002, *Financial Engineering and Computation* (Cambridge, UK: Cambridge University Press).[3]