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

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

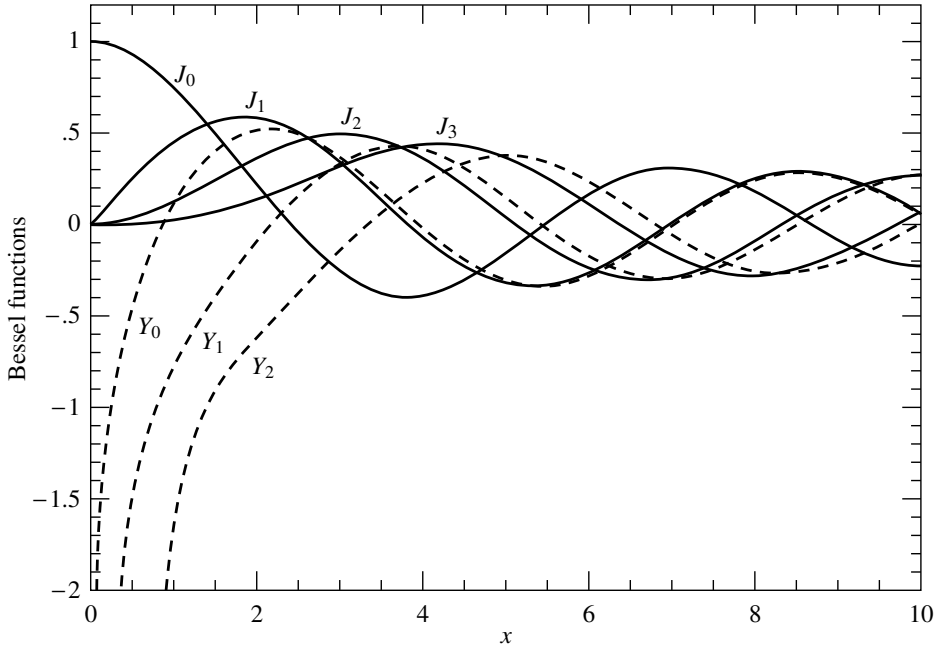

Figure 6.5.1. Bessel functions $J_0(x)$ through $J_3(x)$ and $Y_0(x)$ through $Y_2(x)$.

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,

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

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));
        }
    }
```

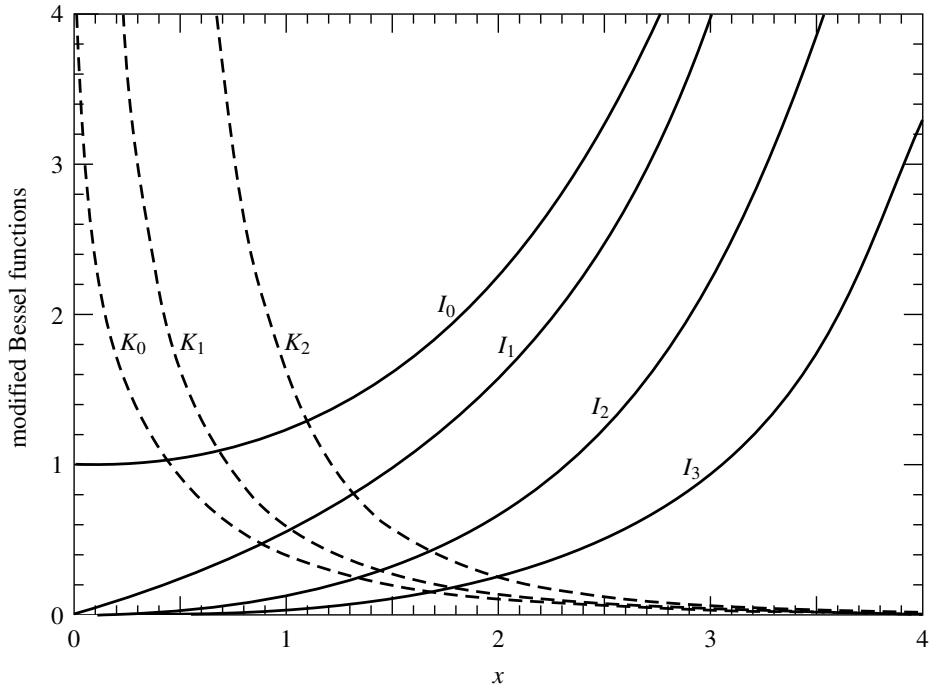

Figure 6.5.2. Modified Bessel functions $I_0(x)$ through $I_3(x)$, and $K_0(x)$ through $K_2(x)$.

```
    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,

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

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 Kn(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:

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

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 In(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://numerical.recipes/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://numerical.recipes/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

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

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 $x \lesssim x_{\rm tp}$,