6.5 Bessel Functions of Integer Order

This section and the next one present practical algorithms for computing various kinds of Bessel functions of integer order. In §6.7 we deal with fractional order. In fact, the more complicated routines for fractional order work fine for integer order too. For integer order, however, the routines in this section (and §6.6) are simpler and faster. Their only drawback is that they are limited by the precision of the underlying rational approximations. For full double precision, it is best to work with the routines for fractional order in §6.7.

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 $0 < x \ll \nu$

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

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

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

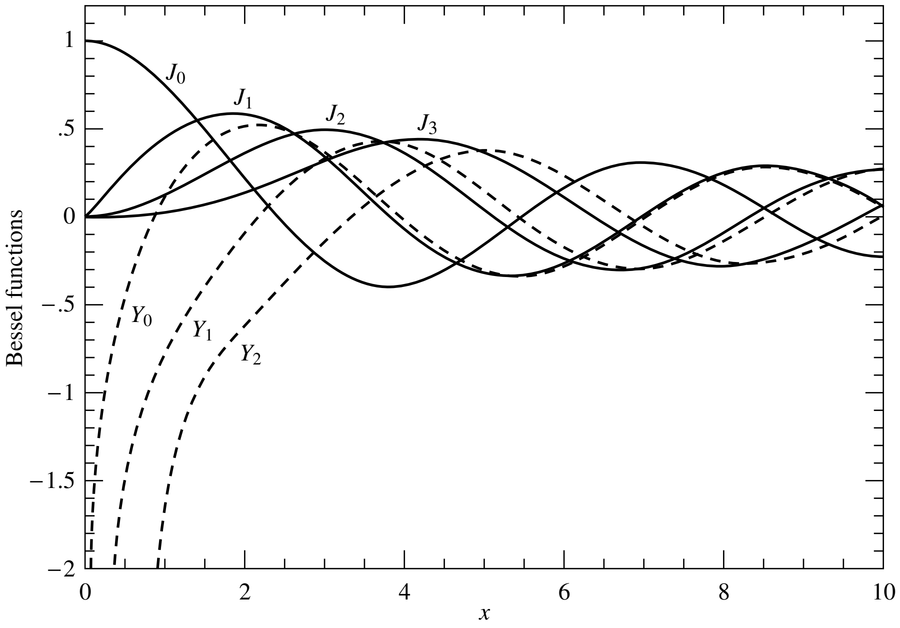

Figure 6.5.1. Bessel functions $J_0(x)$ through $J_3(x)$ and $Y_0(x)$ through $Y_2(x)$.

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

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

and

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

As already mentioned in §5.5, 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 ($n = 0, 1$)

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

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

where

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

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.

Coefficients of the various rational functions and polynomials are given by Hart [1], for various levels of desired accuracy. A straightforward implementation is

```
#include <math.h>

float bessj0(float x)
Returns the Bessel function J0(x) for any real x.
{
    float ax,z;
    double xx,y,ans,ans1,ans2;          Accumulate polynomials in double precision.

    if ((ax=fabs(x)) < 8.0) {           Direct rational function fit.
        y=x*x;
        ans1=57568490574.0+y*(-13362590354.0+y*(651619640.7
            +y*(-11214424.18+y*(77392.33017+y*(-184.9052456)))));
        ans2=57568490411.0+y*(1029532985.0+y*(9494680.718
            +y*(59272.64853+y*(267.8532712+y*1.0))));
        ans=ans1/ans2;
    } else {                            Fitting function (6.5.9).
        z=8.0/ax;
        y=z*z;
        xx=ax-0.785398164;
        ans1=1.0+y*(-0.1098628627e-2+y*(0.2734510407e-4
            +y*(-0.2073370639e-5+y*0.2093887211e-6)));
        ans2 = -0.1562499995e-1+y*(0.1430488765e-3
            +y*(-0.6911147651e-5+y*(0.7621095161e-6
            -y*0.934935152e-7)));
        ans=sqrt(0.636619772/ax)*(cos(xx)*ans1-z*sin(xx)*ans2);
    }
    return ans;
}
```

```
#include <math.h>

float bessy0(float x)
Returns the Bessel function Y0(x) for positive x.
{
    float bessj0(float x);
    float z;
    double xx,y,ans,ans1,ans2;          Accumulate polynomials in double precision.

    if (x < 8.0) {                      Rational function approximation of (6.5.8).
        y=x*x;
        ans1 = -2957821389.0+y*(7062834065.0+y*(-512359803.6
            +y*(10879881.29+y*(-86327.92757+y*228.4622733))));
        ans2=40076544269.0+y*(745249964.8+y*(7189466.438
            +y*(47447.26470+y*(226.1030244+y*1.0))));
        ans=(ans1/ans2)+0.636619772*bessj0(x)*log(x);
    } else {                            Fitting function (6.5.10).
```

```
        z=8.0/x;
        y=z*z;
        xx=x-0.785398164;
        ans1=1.0+y*(-0.1098628627e-2+y*(0.2734510407e-4
            +y*(-0.2073370639e-5+y*0.2093887211e-6)));
        ans2 = -0.1562499995e-1+y*(0.1430488765e-3
            +y*(-0.6911147651e-5+y*(0.7621095161e-6
            +y*(-0.934945152e-7))));
        ans=sqrt(0.636619772/x)*(sin(xx)*ans1+z*cos(xx)*ans2);
    }
    return ans;
}
```

```
#include <math.h>

float bessj1(float x)
Returns the Bessel function J1(x) for any real x.
{
    float ax,z;
    double xx,y,ans,ans1,ans2;          Accumulate polynomials in double precision.

    if ((ax=fabs(x)) < 8.0) {           Direct rational approximation.
        y=x*x;
        ans1=x*(72362614232.0+y*(-7895059235.0+y*(242396853.1
            +y*(-2972611.439+y*(15704.48260+y*(-30.16036606))))));
        ans2=144725228442.0+y*(2300535178.0+y*(18583304.74
            +y*(99447.43394+y*(376.9991397+y*1.0))));
        ans=ans1/ans2;
    } else {                            Fitting function (6.5.9).
        z=8.0/ax;
        y=z*z;
        xx=ax-2.356194491;
        ans1=1.0+y*(0.183105e-2+y*(-0.3516396496e-4
            +y*(0.2457520174e-5+y*(-0.240337019e-6))));
        ans2=0.04687499995+y*(-0.2002690873e-3
            +y*(0.8449199096e-5+y*(-0.88228987e-6
            +y*0.105787412e-6)));
        ans=sqrt(0.636619772/ax)*(cos(xx)*ans1-z*sin(xx)*ans2);
        if (x < 0.0) ans = -ans;
    }
    return ans;
}
```

```
#include <math.h>

float bessy1(float x)
Returns the Bessel function Y1(x) for positive x.
{
    float bessj1(float x);
    float z;
    double xx,y,ans,ans1,ans2;          Accumulate polynomials in double precision.

    if (x < 8.0) {                      Rational function approximation of (6.5.8).
        y=x*x;
        ans1=x*(-0.4900604943e13+y*(0.1275274390e13
            +y*(-0.5153438139e11+y*(0.7349264551e9
            +y*(-0.4237922726e7+y*0.8511937935e4)))));
        ans2=0.2499580570e14+y*(0.4244419664e12
            +y*(0.3733650367e10+y*(0.2245904002e8
            +y*(0.1020426050e6+y*(0.3549632885e3+y)))));
```

```
        ans=(ans1/ans2)+0.636619772*(bessj1(x)*log(x)-1.0/x);
    } else {                                Fitting function (6.5.10).
        z=8.0/x;
        y=z*z;
        xx=x-2.356194491;
        ans1=1.0+y*(0.183105e-2+y*(-0.3516396496e-4
            +y*(0.2457520174e-5+y*(-0.240337019e-6))));
        ans2=0.04687499995+y*(-0.2002690873e-3
            +y*(0.8449199096e-5+y*(-0.88228987e-6
            +y*0.105787412e-6)));
        ans=sqrt(0.636619772/x)*(sin(xx)*ans1+z*cos(xx)*ans2);
    }
    return ans;
}
```

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:

```
float bessy(int n, float x)
Returns the Bessel function Yn(x) for positive x and n ≥ 2.
{
    float bessy0(float x);
    float bessy1(float x);
    void nrerror(char error_text[]);
    int j;
    float by,bym,byp,tox;

    if (n < 2) nrerror("Index n less than 2 in bessy");
    tox=2.0/x;
    by=bessy1(x);                   Starting values for the recurrence.
    bym=bessy0(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 call to `bessy1` and `bessy0` (which generate a call to each of `bessj1` and `bessj0`), plus $O(\mathtt{n})$ operations in the recurrence.

As 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 which 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 5.5.16), 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 (5.5.16) 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 [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.

```
#include <math.h>
#define ACC 40.0                          Make larger to increase accuracy.
#define BIGNO 1.0e10
#define BIGNI 1.0e-10

float bessj(int n, float x)
Returns the Bessel function Jn(x) for any real x and n ≥ 2.
{
    float bessj0(float x);
    float bessj1(float x);
    void nrerror(char error_text[]);
    int j,jsum,m;
    float ax,bj,bjm,bjp,sum,tox,ans;

    if (n < 2) nrerror("Index n less than 2 in bessj");
    ax=fabs(x);
    if (ax == 0.0)
        return 0.0;
    else if (ax > (float) n) {            Upwards recurrence from J0 and J1.
        tox=2.0/ax;
        bjm=bessj0(ax);
        bj=bessj1(ax);
        for (j=1;j<n;j++) {
            bjp=j*tox*bj-bjm;
            bjm=bj;
            bj=bjp;
        }
        ans=bj;
    } else {                              Downwards recurrence from an even m here com-
        tox=2.0/ax;                           puted.
        m=2*((n+(int) sqrt(ACC*n))/2);
        jsum=0;                           jsum will alternate between 0 and 1; when it is
        bjp=ans=sum=0.0;                      1, we accumulate in sum the even terms in
        bj=1.0;                               (5.5.16).
        for (j=m;j>0;j--) {               The downward recurrence.
            bjm=j*tox*bj-bjp;
            bjp=bj;
            bj=bjm;
            if (fabs(bj) > BIGNO) {       Renormalize to prevent overflows.
                bj *= BIGNI;
                bjp *= BIGNI;
                ans *= BIGNI;
                sum *= BIGNI;
            }
            if (jsum) sum += bj;          Accumulate the sum.
            jsum=!jsum;                   Change 0 to 1 or vice versa.
            if (j == n) ans=bjp;          Save the unnormalized answer.
        }
        sum=2.0*sum-bj;                   Compute (5.5.16)
        ans /= sum;                       and use it to normalize the answer.
    }
    return x < 0.0 && (n & 1) ? -ans : ans;
}
```

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapter 9.

Hart, J.F., et al. 1968, *Computer Approximations* (New York: Wiley), §6.8, p. 141. [1]

6.6 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

$$\begin{aligned} I_n(x) &= (-i)^n J_n(ix) \\ K_n(x) &= \frac{\pi}{2} i^{n+1}[J_n(ix) + iY_n(ix)] \end{aligned} \tag{6.6.1}$$

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 argument

$$\begin{aligned} I_n(x) &\approx \frac{1}{n!}\left(\frac{x}{2}\right)^n \qquad n \geq 0 \\ K_0(x) &\approx -\ln(x) \\ K_n(x) &\approx \frac{(n-1)!}{2}\left(\frac{x}{2}\right)^{-n} \qquad n > 0 \end{aligned} \tag{6.6.2}$$

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,

$$\begin{aligned} I_n(x) &\approx \frac{1}{\sqrt{2\pi x}}\exp(x) \\ K_n(x) &\approx \frac{\pi}{\sqrt{2\pi x}}\exp(-x) \end{aligned} \tag{6.6.3}$$

The modified functions evidently have exponential rather than sinusoidal behavior for large arguments (see Figure 6.6.1). The smoothness of the modified Bessel functions, once the exponential factor is removed, makes a simple polynomial approximation of a few terms quite suitable for the functions I_0, I_1, K_0, and K_1. The following routines, based on polynomial coefficients given by Abramowitz and Stegun [1], evaluate these four functions, and will provide the basis for upward recursion for $n > 1$ when $x > n$.