Chapter 4. Integration of Functions

4.0 Introduction

Numerical integration, which is also called *quadrature*, has a history extending back to the invention of calculus and before. The fact that integrals of elementary functions could not, in general, be computed analytically, while derivatives *could* be, served to give the field a certain panache, and to set it a cut above the arithmetic drudgery of numerical analysis during the whole of the 18th and 19th centuries.

With the invention of automatic computing, quadrature became just one numerical task among many, and not a very interesting one at that. Automatic computing, even the most primitive sort involving desk calculators and rooms full of "computers" (that were, until the 1950s, people rather than machines), opened to feasibility the much richer field of numerical integration of differential equations. Quadrature is merely the simplest special case: The evaluation of the integral

$$I = \int_a^b f(x)dx \tag{4.0.1}$$

is precisely equivalent to solving for the value $I \equiv y(b)$ the differential equation

$$\frac{dy}{dx} = f(x) \tag{4.0.2}$$

with the boundary condition

$$y(a) = 0 \tag{4.0.3}$$

Chapter 16 of this book deals with the numerical integration of differential equations. In that chapter, much emphasis is given to the concept of "variable" or "adaptive" choices of stepsize. We will not, therefore, develop that material here. If the function that you propose to integrate is sharply concentrated in one or more peaks, or if its shape is not readily characterized by a single length-scale, then it is likely that you should cast the problem in the form of (4.0.2)–(4.0.3) and use the methods of Chapter 16.

The quadrature methods in this chapter are based, in one way or another, on the obvious device of adding up the value of the integrand at a sequence of abscissas within the range of integration. The game is to obtain the integral as accurately as possible with the smallest number of function evaluations of the integrand. Just as in the case of interpolation (Chapter 3), one has the freedom to choose methods

of various *orders*, with higher order sometimes, but not always, giving higher accuracy. "Romberg integration," which is discussed in §4.3, is a general formalism for making use of integration methods of a variety of different orders, and we recommend it highly.

Apart from the methods of this chapter and of Chapter 16, there are yet other methods for obtaining integrals. One important class is based on function approximation. We discuss explicitly the integration of functions by Chebyshev approximation ("Clenshaw-Curtis" quadrature) in §5.9. Although not explicitly discussed here, you ought to be able to figure out how to do *cubic spline quadrature* using the output of the routine `spline` in §3.3. (Hint: Integrate equation 3.3.3 over x analytically. See [1].)

Some integrals related to Fourier transforms can be calculated using the fast Fourier transform (FFT) algorithm. This is discussed in §13.9.

Multidimensional integrals are another whole multidimensional bag of worms. Section 4.6 is an introductory discussion in this chapter; the important technique of *Monte-Carlo integration* is treated in Chapter 7.

CITED REFERENCES AND FURTHER READING:

Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley), Chapter 2.

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley), Chapter 7.

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 4.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 3.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), Chapter 4.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §7.4.

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

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §5.2, p. 89. [1]

Davis, P., and Rabinowitz, P. 1984, *Methods of Numerical Integration*, 2nd ed. (Orlando, FL: Academic Press).

4.1 Classical Formulas for Equally Spaced Abscissas

Where would any book on numerical analysis be without Mr. Simpson and his "rule"? The classical formulas for integrating a function whose value is known at equally spaced steps have a certain elegance about them, and they are redolent with historical association. Through them, the modern numerical analyst communes with the spirits of his or her predecessors back across the centuries, as far as the time of Newton, if not farther. Alas, times *do* change; with the exception of two of the most modest formulas ("extended trapezoidal rule," equation 4.1.11, and "extended

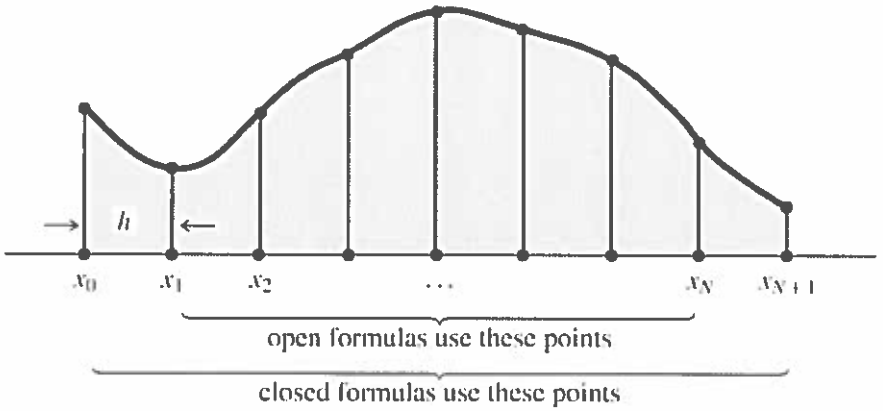

Figure 4.1.1. Quadrature formulas with equally spaced abscissas compute the integral of a function between x_0 and x_{N+1}. Closed formulas evaluate the function on the boundary points, while open formulas refrain from doing so (useful if the evaluation algorithm breaks down on the boundary points).

midpoint rule," equation 4.1.19, see §4.2), the classical formulas are almost entirely useless. They are museum pieces, but beautiful ones.

Some notation: We have a sequence of abscissas, denoted $x_0, x_1, \ldots, x_N$, x_{N+1} which are spaced apart by a constant step h,

$$x_i = x_0 + ih \qquad i = 0, 1, \ldots, N+1 \tag{4.1.1}$$

A function $f(x)$ has known values at the x_i's,

$$f(x_i) \equiv f_i \tag{4.1.2}$$

We want to integrate the function $f(x)$ between a lower limit a and an upper limit b, where a and b are each equal to one or the other of the x_i's. An integration formula that uses the value of the function at the endpoints, $f(a)$ or $f(b)$, is called a *closed* formula. Occasionally, we want to integrate a function whose value at one or both endpoints is difficult to compute (e.g., the computation of f goes to a limit of zero over zero there, or worse yet has an integrable singularity there). In this case we want an *open* formula, which estimates the integral using only x_i's strictly *between* a and b (see Figure 4.1.1).

The basic building blocks of the classical formulas are rules for integrating a function over a small number of intervals. As that number increases, we can find rules that are exact for polynomials of increasingly high order. (Keep in mind that higher order does not always imply higher accuracy in real cases.) A sequence of such closed formulas is now given.

Closed Newton-Cotes Formulas

Trapezoidal rule:

$$\int_{x_1}^{x_2} f(x)dx = h\left[\frac{1}{2}f_1 + \frac{1}{2}f_2\right] \quad + O(h^3 f'') \tag{4.1.3}$$

Here the error term $O(\ )$ signifies that the true answer differs from the estimate by an amount that is the product of some numerical coefficient times h^3 times the value

of the function's second derivative somewhere in the interval of integration. The coefficient is knowable, and it can be found in all the standard references on this subject. The point at which the second derivative is to be evaluated is, however, unknowable. If we knew it, we could evaluate the function there and have a higher-order method! Since the product of a knowable and an unknowable is unknowable, we will streamline our formulas and write only $O(\ )$, instead of the coefficient.

Equation (4.1.3) is a two-point formula (x_1 and x_2). It is exact for polynomials up to and including degree 1, i.e., $f(x) = x$. One anticipates that there is a three-point formula exact up to polynomials of degree 2. This is true; moreover, by a cancellation of coefficients due to left-right symmetry of the formula, the three-point formula is exact for polynomials up to and including degree 3, i.e., $f(x) = x^3$:

Simpson's rule:

$$\int_{x_1}^{x_3} f(x)dx = h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{1}{3}f_3\right] \quad + O(h^5 f^{(4)}) \tag{4.1.4}$$

Here $f^{(4)}$ means the fourth derivative of the function f evaluated at an unknown place in the interval. Note also that the formula gives the integral over an interval of size $2h$, so the coefficients add up to 2.

There is no lucky cancellation in the four-point formula, so it is also exact for polynomials up to and including degree 3.

Simpson's $\frac{3}{8}$ rule:

$$\int_{x_1}^{x_4} f(x)dx = h\left[\frac{3}{8}f_1 + \frac{9}{8}f_2 + \frac{9}{8}f_3 + \frac{3}{8}f_4\right] \quad + O(h^5 f^{(4)}) \tag{4.1.5}$$

The five-point formula again benefits from a cancellation:

Bode's rule:

$$\int_{x_1}^{x_5} f(x)dx = h\left[\frac{14}{45}f_1 + \frac{64}{45}f_2 + \frac{24}{45}f_3 + \frac{64}{45}f_4 + \frac{14}{45}f_5\right] \quad + O(h^7 f^{(6)}) \tag{4.1.6}$$

This is exact for polynomials up to and including degree 5.

At this point the formulas stop being named after famous personages, so we will not go any further. Consult [1] for additional formulas in the sequence.

Extrapolative Formulas for a Single Interval

We are going to depart from historical practice for a moment. Many texts would give, at this point, a sequence of "Newton-Cotes Formulas of Open Type." Here is an example:

$$\int_{x_0}^{x_5} f(x)dx = h\left[\frac{55}{24}f_1 + \frac{5}{24}f_2 + \frac{5}{24}f_3 + \frac{55}{24}f_4\right] \quad + O(h^5 f^{(4)})$$

Notice that the integral from $a = x_0$ to $b = x_5$ is estimated, using only the interior points x_1, x_2, x_3, x_4. In our opinion, formulas of this type are not useful for the reasons that (i) they cannot usefully be strung together to get "extended" rules, as we are about to do with the closed formulas, and (ii) for all other possible uses they are dominated by the Gaussian integration formulas which we will introduce in §4.5.

Instead of the Newton-Cotes open formulas, let us set out the formulas for estimating the integral in the single interval from x_0 to x_1, using values of the function f at $x_1, x_2, \ldots$. These will be useful building blocks for the "extended" open formulas.

$$\int_{x_0}^{x_1} f(x)dx = h[f_1] \quad + O(h^2 f') \tag{4.1.7}$$

$$\int_{x_0}^{x_1} f(x)dx = h\left[\frac{3}{2}f_1 - \frac{1}{2}f_2\right] \quad + O(h^3 f'') \tag{4.1.8}$$

$$\int_{x_0}^{x_1} f(x)dx = h\left[\frac{23}{12}f_1 - \frac{16}{12}f_2 + \frac{5}{12}f_3\right] \quad + O(h^4 f^{(3)}) \tag{4.1.9}$$

$$\int_{x_0}^{x_1} f(x)dx = h\left[\frac{55}{24}f_1 - \frac{59}{24}f_2 + \frac{37}{24}f_3 - \frac{9}{24}f_4\right] \quad + O(h^5 f^{(4)}) \tag{4.1.10}$$

Perhaps a word here would be in order about how formulas like the above can be derived. There are elegant ways, but the most straightforward is to write down the basic form of the formula, replacing the numerical coefficients with unknowns, say p, q, r, s. Without loss of generality take $x_0 = 0$ and $x_1 = 1$, so $h = 1$. Substitute in turn for $f(x)$ (and for f_1, f_2, f_3, f_4) the functions $f(x) = 1$, $f(x) = x$, $f(x) = x^2$, and $f(x) = x^3$. Doing the integral in each case reduces the left-hand side to a number, and the right-hand side to a linear equation for the unknowns p, q, r, s. Solving the four equations produced in this way gives the coefficients.

Extended Formulas (Closed)

If we use equation (4.1.3) $N - 1$ times, to do the integration in the intervals $(x_1, x_2), (x_2, x_3), \ldots, (x_{N-1}, x_N)$, and then add the results, we obtain an "extended" or "composite" formula for the integral from x_1 to x_N.

Extended trapezoidal rule:

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{1}{2}f_1 + f_2 + f_3 + \cdots + f_{N-1} + \frac{1}{2}f_N\right] + O\left(\frac{(b-a)^3 f''}{N^2}\right) \tag{4.1.11}$$

Here we have written the error estimate in terms of the interval $b - a$ and the number of points N instead of in terms of h. This is clearer, since one is usually holding a and b fixed and wanting to know (e.g.) how much the error will be decreased

by taking twice as many steps (in this case, it is by a factor of 4). In subsequent equations we will show *only* the scaling of the error term with the number of steps.

For reasons that will not become clear until §4.2, equation (4.1.11) is in fact the most important equation in this section, the basis for most practical quadrature schemes.

The *extended formula of order* $1/N^3$ is:

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{5}{12}f_1 + \frac{13}{12}f_2 + f_3 + f_4 + \cdots + f_{N-2} + \frac{13}{12}f_{N-1} + \frac{5}{12}f_N\right] + O\left(\frac{1}{N^3}\right) \tag{4.1.12}$$

(We will see in a moment where this comes from.)

If we apply equation (4.1.4) to successive, nonoverlapping *pairs* of intervals, we get the *extended Simpson's rule:*

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{2}{3}f_3 + \frac{4}{3}f_4 + \cdots + \frac{2}{3}f_{N-2} + \frac{4}{3}f_{N-1} + \frac{1}{3}f_N\right] + O\left(\frac{1}{N^4}\right) \tag{4.1.13}$$

Notice that the 2/3, 4/3 alternation continues throughout the interior of the evaluation. Many people believe that the wobbling alternation somehow contains deep information about the integral of their function that is not apparent to mortal eyes. In fact, the alternation is an artifact of using the building block (4.1.4). Another extended formula with the same order as Simpson's rule is

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{3}{8}f_1 + \frac{7}{6}f_2 + \frac{23}{24}f_3 + f_4 + f_5 + \cdots + f_{N-4} + f_{N-3} + \frac{23}{24}f_{N-2} + \frac{7}{6}f_{N-1} + \frac{3}{8}f_N\right] + O\left(\frac{1}{N^4}\right) \tag{4.1.14}$$

This equation is constructed by fitting cubic polynomials through successive groups of four points; we defer details to §18.3, where a similar technique is used in the solution of integral equations. We can, however, tell you where equation (4.1.12) came from. It is Simpson's extended rule, averaged with a modified version of itself in which the first and last step are done with the trapezoidal rule (4.1.3). The trapezoidal step is *two* orders lower than Simpson's rule; however, its contribution to the integral goes down as an additional power of N (since it is used only twice, not N times). This makes the resulting formula of degree *one* less than Simpson.

Extended Formulas (Open and Semi-open)

We can construct open and semi-open extended formulas by adding the closed formulas (4.1.11)–(4.1.14), evaluated for the second and subsequent steps, to the extrapolative open formulas for the first step, (4.1.7)–(4.1.10). As discussed immediately above, it is consistent to use an end step that is of one order lower than the (repeated) interior step. The resulting formulas for an interval open at both ends are as follows:

Equations (4.1.7) and (4.1.11) give

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{3}{2}f_2 + f_3 + f_4 + \cdots + f_{N-2} + \frac{3}{2}f_{N-1}\right] + O\left(\frac{1}{N^2}\right) \quad (4.1.15)$$

Equations (4.1.8) and (4.1.12) give

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{23}{12}f_2 + \frac{7}{12}f_3 + f_4 + f_5 + \cdots + f_{N-3} + \frac{7}{12}f_{N-2} + \frac{23}{12}f_{N-1}\right] + O\left(\frac{1}{N^3}\right) \quad (4.1.16)$$

Equations (4.1.9) and (4.1.13) give

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{27}{12}f_2 + 0 + \frac{13}{12}f_4 + \frac{4}{3}f_5 + \cdots + \frac{4}{3}f_{N-4} + \frac{13}{12}f_{N-3} + 0 + \frac{27}{12}f_{N-1}\right] + O\left(\frac{1}{N^4}\right) \quad (4.1.17)$$

The interior points alternate 4/3 and 2/3. If we want to avoid this alternation, we can combine equations (4.1.9) and (4.1.14), giving

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{55}{24}f_2 - \frac{1}{6}f_3 + \frac{11}{8}f_4 + f_5 + f_6 + f_7 + \cdots + f_{N-5} + f_{N-4} + \frac{11}{8}f_{N-3} - \frac{1}{6}f_{N-2} + \frac{55}{24}f_{N-1}\right] + O\left(\frac{1}{N^4}\right) \quad (4.1.18)$$

We should mention in passing another extended open formula, for use where the limits of integration are located halfway between tabulated abscissas. This one is known as the *extended midpoint rule*, and is accurate to the same order as (4.1.15):

$$\int_{x_1}^{x_N} f(x)dx = h[f_{3/2} + f_{5/2} + f_{7/2} + \cdots + f_{N-3/2} + f_{N-1/2}] + O\left(\frac{1}{N^2}\right) \quad (4.1.19)$$

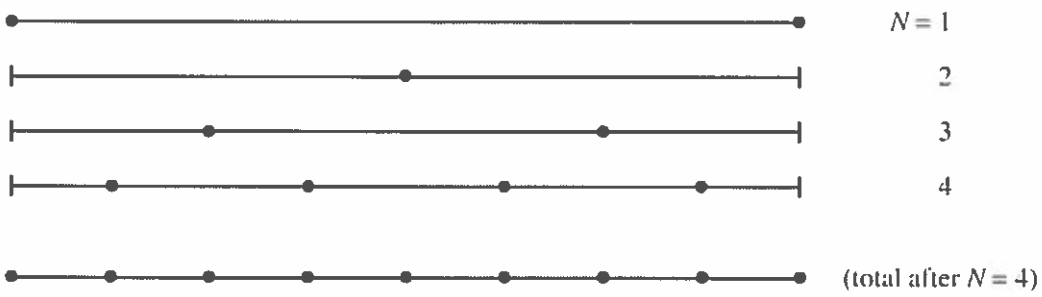

Figure 4.2.1. Sequential calls to the routine `trapzd` incorporate the information from previous calls and evaluate the integrand only at those new points necessary to refine the grid. The bottom line shows the totality of function evaluations after the fourth call. The routine `qsimp`, by weighting the intermediate results, transforms the trapezoid rule into Simpson's rule with essentially no additional overhead.

There are also formulas of higher order for this situation, but we will refrain from giving them.

The *semi-open formulas* are just the obvious combinations of equations (4.1.11)–(4.1.14) with (4.1.15)–(4.1.18), respectively. At the closed end of the integration, use the weights from the former equations; at the open end use the weights from the latter equations. One example should give the idea, the formula with error term decreasing as $1/N^3$ which is closed on the right and open on the left:

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{23}{12}f_2 + \frac{7}{12}f_3 + f_4 + f_5 + \cdots + f_{N-2} + \frac{13}{12}f_{N-1} + \frac{5}{12}f_N\right] + O\left(\frac{1}{N^3}\right) \tag{4.1.20}$$

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), §25.4. [1]

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley), §7.1.

4.2 Elementary Algorithms

Our starting point is equation (4.1.11), the extended trapezoidal rule. There are two facts about the trapezoidal rule which make it the starting point for a variety of algorithms. One fact is rather obvious, while the second is rather "deep."

The obvious fact is that, for a fixed function $f(x)$ to be integrated between fixed limits a and b, one can double the number of intervals in the extended trapezoidal rule without losing the benefit of previous work. The coarsest implementation of the trapezoidal rule is to average the function at its endpoints a and b. The first stage of refinement is to add to this average the value of the function at the halfway point. The second stage of refinement is to add the values at the 1/4 and 3/4 points. And so on (see Figure 4.2.1).

Without further ado we can write a routine with this kind of logic in it:

```
#define FUNC(x) ((*func)(x))

float trapzd(float (*func)(float), float a, float b, int n)
This routine computes the nth stage of refinement of an extended trapezoidal rule. func is input
as a pointer to the function to be integrated between limits a and b, also input. When called with
n=1, the routine returns the crudest estimate of ∫_a^b f(x)dx. Subsequent calls with n=2,3,...
(in that sequential order) will improve the accuracy by adding 2^(n-2) additional interior points.
{
    float x,tnm,sum,del;
    static float s;
    int it,j;

    if (n == 1) {
        return (s=0.5*(b-a)*(FUNC(a)+FUNC(b)));
    } else {
        for (it=1,j=1;j<n-1;j++) it <<= 1;
        tnm=it;
        del=(b-a)/tnm;                  This is the spacing of the points to be added.
        x=a+0.5*del;
        for (sum=0.0,j=1;j<=it;j++,x+=del) sum += FUNC(x);
        s=0.5*(s+(b-a)*sum/tnm);        This replaces s by its refined value.
        return s;
    }
}
```

The above routine (`trapzd`) is a workhorse that can be harnessed in several ways. The simplest and crudest is to integrate a function by the extended trapezoidal rule where you know in advance (we can't imagine how!) the number of steps you want. If you want $2^M + 1$, you can accomplish this by the fragment

```
for(j=1;j<=m+1;j++) s=trapzd(func,a,b,j);
```

with the answer returned as `s`.

Much better, of course, is to refine the trapezoidal rule until some specified degree of accuracy has been achieved:

```
#include <math.h>
#define EPS 1.0e-5
#define JMAX 20

float qtrap(float (*func)(float), float a, float b)
Returns the integral of the function func from a to b. The parameters EPS can be set to the
desired fractional accuracy and JMAX so that 2 to the power JMAX-1 is the maximum allowed
number of steps. Integration is performed by the trapezoidal rule.
{
    float trapzd(float (*func)(float), float a, float b, int n);
    void nrerror(char error_text[]);
    int j;
    float s,olds=0.0;               Initial value of olds is arbitrary.

    for (j=1;j<=JMAX;j++) {
        s=trapzd(func,a,b,j);
        if (j > 5)                  Avoid spurious early convergence.
            if (fabs(s-olds) < EPS*fabs(olds) ||
                (s == 0.0 && olds == 0.0)) return s;
        olds=s;
    }
    nrerror("Too many steps in routine qtrap");
    return 0.0;                     Never get here.
}
```

Unsophisticated as it is, routine `qtrap` is in fact a fairly robust way of doing integrals of functions that are not very smooth. Increased sophistication will usually translate into a higher-order method whose efficiency will be greater only for sufficiently smooth integrands. `qtrap` is the method of choice, e.g., for an integrand which is a function of a variable that is linearly interpolated between measured data points. Be sure that you do not require too stringent an EPS, however: If `qtrap` takes too many steps in trying to achieve your required accuracy, accumulated roundoff errors may start increasing, and the routine may never converge. A value 10^{-6} is just on the edge of trouble for most 32-bit machines; it is achievable when the convergence is moderately rapid, but not otherwise.

We come now to the "deep" fact about the extended trapezoidal rule, equation (4.1.11). It is this: The error of the approximation, which begins with a term of order $1/N^2$, is in fact *entirely even* when expressed in powers of $1/N$. This follows directly from the *Euler-Maclaurin Summation Formula*.

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{1}{2}f_1 + f_2 + f_3 + \cdots + f_{N-1} + \frac{1}{2}f_N\right] - \frac{B_2 h^2}{2!}(f'_N - f'_1) - \cdots - \frac{B_{2k}h^{2k}}{(2k)!}(f_N^{(2k-1)} - f_1^{(2k-1)}) - \cdots \tag{4.2.1}$$

Here B_{2k} is a *Bernoulli number*, defined by the generating function

$$\frac{t}{e^t - 1} = \sum_{n=0}^{\infty} B_n \frac{t^n}{n!} \tag{4.2.2}$$

with the first few even values (odd values vanish except for $B_1 = -1/2$)

$$B_0 = 1 \quad B_2 = \frac{1}{6} \quad B_4 = -\frac{1}{30} \quad B_6 = \frac{1}{42}$$
$$B_8 = -\frac{1}{30} \quad B_{10} = \frac{5}{66} \quad B_{12} = -\frac{691}{2730} \tag{4.2.3}$$

Equation (4.2.1) is not a convergent expansion, but rather only an asymptotic expansion whose error when truncated at any point is always less than twice the magnitude of the first neglected term. The reason that it is not convergent is that the Bernoulli numbers become very large, e.g.,

$$B_{50} = \frac{495057205241079648212477525}{66}$$

The key point is that only even powers of h occur in the error series of (4.2.1). This fact is not, in general, shared by the higher-order quadrature rules in §4.1. For example, equation (4.1.12) has an error series beginning with $O(1/N^3)$, but continuing with all subsequent powers of N: $1/N^4$, $1/N^5$, etc.

Suppose we evaluate (4.1.11) with N steps, getting a result S_N, and then again with $2N$ steps, getting a result S_{2N}. (This is done by any two consecutive calls of

trapzd.) The leading error term in the second evaluation will be 1/4 the size of the error in the first evaluation. Therefore the combination

$$S = \frac{4}{3}S_{2N} - \frac{1}{3}S_N \tag{4.2.4}$$

will cancel out the leading order error term. But there *is* no error term of order $1/N^3$, by (4.2.1). The surviving error is of order $1/N^4$, the same as Simpson's rule. In fact, it should not take long for you to see that (4.2.4) is *exactly* Simpson's rule (4.1.13), alternating 2/3's, 4/3's, and all. This is the preferred method for evaluating that rule, and we can write it as a routine exactly analogous to qtrap above:

```
#include <math.h>
#define EPS 1.0e-6
#define JMAX 20

float qsimp(float (*func)(float), float a, float b)
Returns the integral of the function func from a to b. The parameters EPS can be set to the
desired fractional accuracy and JMAX so that 2 to the power JMAX-1 is the maximum allowed
number of steps. Integration is performed by Simpson's rule.
{
    float trapzd(float (*func)(float), float a, float b, int n);
    void nrerror(char error_text[]);
    int j;
    float s,st,ost=0.0,os=0.0;

    for (j=1;j<=JMAX;j++) {
        st=trapzd(func,a,b,j);
        s=(4.0*st-ost)/3.0;                Compare equation (4.2.4), above.
        if (j > 5)                         Avoid spurious early convergence.
            if (fabs(s-os) < EPS*fabs(os) ||
                (s == 0.0 && os == 0.0)) return s;
        os=s;
        ost=st;
    }
    nrerror("Too many steps in routine qsimp");
    return 0.0;                            Never get here.
}
```

The routine qsimp will in general be more efficient than qtrap (i.e., require fewer function evaluations) when the function to be integrated has a finite 4th derivative (i.e., a continuous 3rd derivative). The combination of qsimp and its necessary workhorse trapzd is a good one for light-duty work.

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §3.3.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §§7.4.1–7.4.2.

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §5.3.

4.3 Romberg Integration

We can view Romberg's method as the natural generalization of the routine qsimp in the last section to integration schemes that are of higher order than Simpson's rule. The basic idea is to use the results from k successive refinements of the extended trapezoidal rule (implemented in trapzd) to remove all terms in the error series up to but not including $O(1/N^{2k})$. The routine qsimp is the case of $k = 2$. This is one example of a very general idea that goes by the name of *Richardson's deferred approach to the limit*: Perform some numerical algorithm for various values of a parameter h, and then extrapolate the result to the continuum limit $h = 0$.

Equation (4.2.4), which subtracts off the leading error term, is a special case of polynomial extrapolation. In the more general Romberg case, we can use Neville's algorithm (see §3.1) to extrapolate the successive refinements to zero stepsize. Neville's algorithm can in fact be coded very concisely within a Romberg integration routine. For clarity of the program, however, it seems better to do the extrapolation by function call to polint, already given in §3.1.

```
#include <math.h>
#define EPS 1.0e-6
#define JMAX 20
#define JMAXP (JMAX+1)
#define K 5
```
Here EPS is the fractional accuracy desired, as determined by the extrapolation error estimate; JMAX limits the total number of steps; K is the number of points used in the extrapolation.

```
float qromb(float (*func)(float), float a, float b)
```
Returns the integral of the function func from a to b. Integration is performed by Romberg's method of order 2K, where, e.g., K=2 is Simpson's rule.
```
{
    void polint(float xa[], float ya[], int n, float x, float *y, float *dy);
    float trapzd(float (*func)(float), float a, float b, int n);
    void nrerror(char error_text[]);
    float ss,dss;
    float s[JMAXP],h[JMAXP+1];          These store the successive trapezoidal approxi-
    int j;                                  mations and their relative stepsizes.

    h[1]=1.0;
    for (j=1;j<=JMAX;j++) {
        s[j]=trapzd(func,a,b,j);
        if (j >= K) {
            polint(&h[j-K],&s[j-K],K,0.0,&ss,&dss);
            if (fabs(dss) <= EPS*fabs(ss)) return ss;
        }
        h[j+1]=0.25*h[j];
        This is a key step: The factor is 0.25 even though the stepsize is decreased by only
        0.5. This makes the extrapolation a polynomial in h^2 as allowed by equation (4.2.1),
        not just a polynomial in h.
    }
    nrerror("Too many steps in routine qromb");
    return 0.0;                         Never get here.
}
```

The routine qromb, along with its required trapzd and polint, is quite powerful for sufficiently smooth (e.g., analytic) integrands, integrated over intervals

which contain no singularities, and where the endpoints are also nonsingular, `qromb`, in such circumstances, takes many, *many* fewer function evaluations than either of the routines in §4.2. For example, the integral

$$\int_0^2 x^4 \log(x + \sqrt{x^2 + 1})dx$$

converges (with parameters as shown above) on the very first extrapolation, after just 5 calls to `trapzd`, while `qsimp` requires 8 calls (8 times as many evaluations of the integrand) and `qtrap` requires 13 calls (making 256 times as many evaluations of the integrand).

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §§3.4–3.5.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §§7.4.1–7.4.2.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §4.10–2.

4.4 Improper Integrals

For our present purposes, an integral will be "improper" if it has any of the following problems:

- its integrand goes to a finite limiting value at finite upper and lower limits, but cannot be evaluated *right on* one of those limits (e.g., $\sin x/x$ at $x = 0$)
- its upper limit is ∞ , or its lower limit is $-\infty$
- it has an integrable singularity at either limit (e.g., $x^{-1/2}$ at $x = 0$)
- it has an integrable singularity at a known place between its upper and lower limits
- it has an integrable singularity at an unknown place between its upper and lower limits

If an integral is infinite (e.g., $\int_1^\infty x^{-1}dx$), or does not exist in a limiting sense (e.g., $\int_{-\infty}^\infty \cos x dx$), we do not call it improper; we call it impossible. No amount of clever algorithmics will return a meaningful answer to an ill-posed problem.

In this section we will generalize the techniques of the preceding two sections to cover the first four problems on the above list. A more advanced discussion of quadrature with integrable singularities occurs in Chapter 18, notably §18.3. The fifth problem, singularity at unknown location, can really only be handled by the use of a variable stepsize differential equation integration routine, as will be given in Chapter 16.

We need a workhorse like the extended trapezoidal rule (equation 4.1.11), but one which is an *open* formula in the sense of §4.1, i.e., does not require the integrand to be evaluated at the endpoints. Equation (4.1.19), the extended midpoint rule, is the best choice. The reason is that (4.1.19) shares with (4.1.11) the "deep" property of

having an error series that is entirely even in h. Indeed there is a formula, not as well known as it ought to be, called the *Second Euler-Maclaurin summation formula*.

$$\int_{x_1}^{x_N} f(x)dx = h[f_{3/2} + f_{5/2} + f_{7/2} + \cdots + f_{N-3/2} + f_{N-1/2}] + \frac{B_2 h^2}{4}(f'_N - f'_1) + \cdots + \frac{B_{2k}h^{2k}}{(2k)!}(1 - 2^{-2k+1})(f_N^{(2k-1)} - f_1^{(2k-1)}) + \cdots \tag{4.4.1}$$

This equation can be derived by writing out (4.2.1) with stepsize h, then writing it out again with stepsize $h/2$, then subtracting the first from twice the second.

It is not possible to double the number of steps in the extended midpoint rule and still have the benefit of previous function evaluations (try it!). However, it is possible to *triple* the number of steps and do so. Shall we do this, or double and accept the loss? On the average, tripling does a factor $\sqrt{3}$ of unnecessary work, since the "right" number of steps for a desired accuracy criterion may in fact fall anywhere in the logarithmic interval implied by tripling. For doubling, the factor is only $\sqrt{2}$, but we lose an extra factor of 2 in being unable to use all the previous evaluations. Since $1.732 < 2 \times 1.414$, it is better to triple.

Here is the resulting routine, which is directly comparable to `trapzd`.

```
#define FUNC(x) ((*func)(x))

float midpnt(float (*func)(float), float a, float b, int n)
This routine computes the nth stage of refinement of an extended midpoint rule. func is input
as a pointer to the function to be integrated between limits a and b, also input. When called with
n=1, the routine returns the crudest estimate of ∫_a^b f(x)dx. Subsequent calls with n=2,3,...
(in that sequential order) will improve the accuracy of s by adding (2/3) × 3^(n-1) additional
interior points. s should not be modified between sequential calls.
{
    float x,tnm,sum,del,ddel;
    static float s;
    int it,j;

    if (n == 1) {
        return (s=(b-a)*FUNC(0.5*(a+b)));
    } else {
        for(it=1,j=1;j<n-1;j++) it *= 3;
        tnm=it;
        del=(b-a)/(3.0*tnm);
        ddel=del+del;                    The added points alternate in spacing between
        x=a+0.5*del;                         del and ddel.
        sum=0.0;
        for (j=1;j<=it;j++) {
            sum += FUNC(x);
            x += ddel;
            sum += FUNC(x);
            x += del;
        }
        s=(s+(b-a)*sum/tnm)/3.0;         The new sum is combined with the old integral
        return s;                            to give a refined integral.
    }
}
```

The routine midpnt can exactly replace trapzd in a driver routine like qtrap (§4.2): one simply changes trapzd(func,a,b,j) to midpnt(func,a,b,j), and perhaps also decreases the parameter JMAX since 3^{JMAX-1} (from step tripling) is a much larger number than 2^{JMAX-1} (step doubling).

The open formula implementation analogous to Simpson's rule (qsimp in §4.2) substitutes midpnt for trapzd and decreases JMAX as above, but now also changes the extrapolation step to be

```
s=(9.0*st-ost)/8.0;
```

since, when the number of steps is tripled, the error decreases to 1/9th its size, not 1/4th as with step doubling.

Either the modified qtrap or the modified qsimp will fix the first problem on the list at the beginning of this section. Yet more sophisticated is to generalize Romberg integration in like manner:

```
#include <math.h>
#define EPS 1.0e-6
#define JMAX 14
#define JMAXP (JMAX+1)
#define K 5

float qromo(float (*func)(float), float a, float b,
    float (*choose)(float(*)(float), float, float, int))
Romberg integration on an open interval. Returns the integral of the function func from a to b,
using any specified integrating function choose and Romberg's method. Normally choose will
be an open formula, not evaluating the function at the endpoints. It is assumed that choose
triples the number of steps on each call, and that its error series contains only even powers of
the number of steps. The routines midpnt, midinf, midsql, midsqu, midexp, are possible
choices for choose. The parameters have the same meaning as in qromb.
{
    void polint(float xa[], float ya[], int n, float x, float *y, float *dy);
    void nrerror(char error_text[]);
    int j;
    float ss,dss,h[JMAXP+1],s[JMAXP];

    h[1]=1.0;
    for (j=1;j<=JMAX;j++) {
        s[j]=(*choose)(func,a,b,j);
        if (j >= K) {
            polint(&h[j-K],&s[j-K],K,0.0,&ss,&dss);
            if (fabs(dss) <= EPS*fabs(ss)) return ss;
        }
        h[j+1]=h[j]/9.0;          This is where the assumption of step tripling and an even
    }                                 error series is used.
    nrerror("Too many steps in routing qromo");
    return 0.0;                   Never get here.
}
```

Don't be put off by qromo's complicated ANSI declaration. A typical invocation (integrating the Bessel function $Y_0(x)$ from 0 to 2) is simply

```
#include "nr.h"
float answer;
...
answer=qromo(bessy0,0.0,2.0,midpnt);
```

The differences between qromo and qromb (§4.3) are so slight that it is perhaps gratuitous to list qromo in full. It, however, is an excellent driver routine for solving all the other problems of improper integrals in our first list (except the intractable fifth), as we shall now see.

The basic trick for improper integrals is to make a change of variables to eliminate the singularity, or to map an infinite range of integration to a finite one. For example, the identity

$$\int_a^b f(x)dx = \int_{1/b}^{1/a} \frac{1}{t^2} f\left(\frac{1}{t}\right) dt \qquad ab > 0 \tag{4.4.2}$$

can be used with *either* $b \to \infty$ and a positive, *or* with $a \to -\infty$ and b negative, and works for any function which decreases towards infinity faster than $1/x^2$.

You can make the change of variable implied by (4.4.2) either analytically and then use (e.g.) qromo and midpnt to do the numerical evaluation, *or* you can let the numerical algorithm make the change of variable for you. We prefer the latter method as being more transparent to the user. To implement equation (4.4.2) we simply write a modified version of midpnt, called midinf, which allows b to be infinite (or, more precisely, a very large number on your particular machine, such as 1×10^{30}), or a to be negative and infinite.

```
#define FUNC(x) ((*funk)(1.0/(x))/((x)*(x)))        Effects the change of variable.

float midinf(float (*funk)(float), float aa, float bb, int n)
This routine is an exact replacement for midpnt, i.e., returns the nth stage of refinement of
the integral of funk from aa to bb, except that the function is evaluated at evenly spaced
points in 1/x rather than in x. This allows the upper limit bb to be as large and positive as
the computer allows, or the lower limit aa to be as large and negative, but not both. aa and
bb must have the same sign.
{
    float x,tnm,sum,del,ddel,b,a;
    static float s;
    int it,j;

    b=1.0/aa;                          These two statements change the limits of integration.
    a=1.0/bb;
    if (n == 1) {                      From this point on, the routine is identical to midpnt.
        return (s=(b-a)*FUNC(0.5*(a+b)));
    } else {
        for(it=1,j=1;j<n-1;j++) it *= 3;
        tnm=it;
        del=(b-a)/(3.0*tnm);
        ddel=del+del;
        x=a+0.5*del;
        sum=0.0;
        for (j=1;j<=it;j++) {
            sum += FUNC(x);
            x += ddel;
            sum += FUNC(x);
            x += del;
        }
        return (s=(s+(b-a)*sum/tnm)/3.0);
    }
}
```

If you need to integrate from a negative lower limit to positive infinity, you do this by breaking the integral into two pieces at some positive value, for example,

```
answer=qromo(funk,-5.0,2.0,midpnt)+qromo(funk,2.0,1.0e30,midinf);
```

Where should you choose the breakpoint? At a sufficiently large positive value so that the function `funk` is at least beginning to approach its asymptotic decrease to zero value at infinity. The polynomial extrapolation implicit in the second call to `qromo` deals with a polynomial in $1/x$, not in x.

To deal with an integral that has an integrable power-law singularity at its lower limit, one also makes a change of variable. If the integrand diverges as $(x-a)^{-\gamma}$, $0 \le \gamma < 1$, near $x = a$, use the identity

$$\int_a^b f(x)dx = \frac{1}{1-\gamma}\int_0^{(b-a)^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}} f(t^{\frac{1}{1-\gamma}} + a)dt \qquad (b > a) \tag{4.4.3}$$

If the singularity is at the upper limit, use the identity

$$\int_a^b f(x)dx = \frac{1}{1-\gamma}\int_0^{(b-a)^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}} f(b - t^{\frac{1}{1-\gamma}})dt \qquad (b > a) \tag{4.4.4}$$

If there is a singularity at both limits, divide the integral at an interior breakpoint as in the example above.

Equations (4.4.3) and (4.4.4) are particularly simple in the case of inverse square-root singularities, a case that occurs frequently in practice:

$$\int_a^b f(x)dx = \int_0^{\sqrt{b-a}} 2t f(a + t^2)dt \qquad (b > a) \tag{4.4.5}$$

for a singularity at a, and

$$\int_a^b f(x)dx = \int_0^{\sqrt{b-a}} 2t f(b - t^2)dt \qquad (b > a) \tag{4.4.6}$$

for a singularity at b. Once again, we can implement these changes of variable transparently to the user by defining substitute routines for `midpnt` which make the change of variable automatically:

```
#include <math.h>

#define FUNC(x) (2.0*(x)*(*funk)(aa+(x)*(x)))

float midsql(float (*funk)(float), float aa, float bb, int n)
This routine is an exact replacement for midpnt, except that it allows for an inverse square-root
singularity in the integrand at the lower limit aa.
{
    float x,tnm,sum,del,ddel,a,b;
    static float s;
    int it,j;

    b=sqrt(bb-aa);
    a=0.0;
    if (n == 1) {
The rest of the routine is exactly like midpnt and is omitted.
```

Similarly,

```
#include <math.h>

#define FUNC(x) (2.0*(x)*(*funk)(bb-(x)*(x)))

float midsqu(float (*funk)(float), float aa, float bb, int n)
This routine is an exact replacement for midpnt, except that it allows for an inverse square-root
singularity in the integrand at the upper limit bb.
{
    float x,tnm,sum,del,ddel,a,b;
    static float s;
    int it,j;

    b=sqrt(bb-aa);
    a=0.0;
    if (n == 1) {
The rest of the routine is exactly like midpnt and is omitted.
```

One last example should suffice to show how these formulas are derived in general. Suppose the upper limit of integration is infinite, and the integrand falls off exponentially. Then we want a change of variable that maps $e^{-x}dx$ into $(\pm)dt$ (with the sign chosen to keep the upper limit of the new variable larger than the lower limit). Doing the integration gives by inspection

$$t = e^{-x} \qquad \text{or} \qquad x = -\log t \tag{4.4.7}$$

so that

$$\int_{x=a}^{x=\infty} f(x)dx = \int_{t=0}^{t=e^{-a}} f(-\log t)\frac{dt}{t} \tag{4.4.8}$$

The user-transparent implementation would be

```
#include <math.h>

#define FUNC(x) ((*funk)(-log(x))/(x))

float midexp(float (*funk)(float), float aa, float bb, int n)
This routine is an exact replacement for midpnt, except that bb is assumed to be infinite
(value passed not actually used). It is assumed that the function funk decreases exponentially
rapidly at infinity.
{
    float x,tnm,sum,del,ddel,a,b;
    static float s;
    int it,j;

    b=exp(-aa);
    a=0.0;
    if (n == 1) {
The rest of the routine is exactly like midpnt and is omitted.
```

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 4.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §7.4.3, p. 294.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §3.7, p. 152.

4.5 Gaussian Quadratures and Orthogonal Polynomials

In the formulas of §4.1, the integral of a function was approximated by the sum of its functional values at a set of equally spaced points, multiplied by certain aptly chosen weighting coefficients. We saw that as we allowed ourselves more freedom in choosing the coefficients, we could achieve integration formulas of higher and higher order. The idea of *Gaussian quadratures* is to give ourselves the freedom to choose not only the weighting coefficients, but also the location of the abscissas at which the function is to be evaluated: They will no longer be equally spaced. Thus, we will have *twice* the number of degrees of freedom at our disposal; it will turn out that we can achieve Gaussian quadrature formulas whose order is, essentially, twice that of the Newton-Cotes formula with the same number of function evaluations.

Does this sound too good to be true? Well, in a sense it is. The catch is a familiar one, which cannot be overemphasized: High order is not the same as high accuracy. High order translates to high accuracy only when the integrand is very smooth, in the sense of being "well-approximated by a polynomial."

There is, however, one additional feature of Gaussian quadrature formulas that adds to their usefulness: We can arrange the choice of weights and abscissas to make the integral exact for a class of integrands "polynomials times some known function $W(x)$" rather than for the usual class of integrands "polynomials." The function $W(x)$ can then be chosen to remove integrable singularities from the desired integral. Given $W(x)$, in other words, and given an integer N, we can find a set of weights w_j and abscissas x_j such that the approximation

$$\int_a^b W(x)f(x)dx \approx \sum_{j=1}^{N} w_j f(x_j) \tag{4.5.1}$$

is exact if $f(x)$ is a polynomial. For example, to do the integral

$$\int_{-1}^{1} \frac{\exp(-\cos^2 x)}{\sqrt{1-x^2}}dx \tag{4.5.2}$$

(not a very natural looking integral, it must be admitted), we might well be interested in a Gaussian quadrature formula based on the choice

$$W(x) = \frac{1}{\sqrt{1-x^2}} \tag{4.5.3}$$

in the interval $(-1, 1)$. (This particular choice is called *Gauss-Chebyshev integration*, for reasons that will become clear shortly.)

Notice that the integration formula (4.5.1) can also be written with the weight function $W(x)$ not overtly visible: Define $g(x) \equiv W(x)f(x)$ and $v_j \equiv w_j/W(x_j)$. Then (4.5.1) becomes

$$\int_a^b g(x)dx \approx \sum_{j=1}^{N} v_j g(x_j) \tag{4.5.4}$$

Where did the function $W(x)$ go? It is lurking there, ready to give high-order accuracy to integrands of the form polynomials times $W(x)$, and ready to *deny* high-order accuracy to integrands that are otherwise perfectly smooth and well-behaved. When you find tabulations of the weights and abscissas for a given $W(x)$, you have to determine carefully whether they are to be used with a formula in the form of (4.5.1), or like (4.5.4).

Here is an example of a quadrature routine that contains the tabulated abscissas and weights for the case $W(x) = 1$ and $N = 10$. Since the weights and abscissas are, in this case, symmetric around the midpoint of the range of integration, there are actually only five distinct values of each:

```
float qgaus(float (*func)(float), float a, float b)
Returns the integral of the function func between a and b, by ten-point Gauss-Legendre integration: the function is evaluated exactly ten times at interior points in the range of integration.
{
    int j;
    float xr,xm,dx,s;
    static float x[]={0.0,0.1488743389,0.4333953941,
        0.6794095682,0.8650633666,0.9739065285};
    static float w[]={0.0,0.2955242247,0.2692667193,
        0.2190863625,0.1494513491,0.0666713443};

    xm=0.5*(b+a);
    xr=0.5*(b-a);
    s=0;
    for (j=1;j<=5;j++) {
        dx=xr*x[j];
        s += w[j]*((*func)(xm+dx)+(*func)(xm-dx));
    }
    return s *= xr;
}
```

The abscissas and weights. First value of each array not used.

Will be twice the average value of the function, since the ten weights (five numbers above each used twice) sum to 2.

Scale the answer to the range of integration.

The above routine illustrates that one can use Gaussian quadratures without necessarily understanding the theory behind them: One just locates tabulated weights and abscissas in a book (e.g., [1] or [2]). However, the theory is very pretty, and it will come in handy if you ever need to construct your own tabulation of weights and abscissas for an unusual choice of $W(x)$. We will therefore give, without any proofs, some useful results that will enable you to do this. Several of the results assume that $W(x)$ does not change sign inside (a, b), which is usually the case in practice.

The theory behind Gaussian quadratures goes back to Gauss in 1814, who used continued fractions to develop the subject. In 1826 Jacobi rederived Gauss's results by means of orthogonal polynomials. The systematic treatment of arbitrary weight functions $W(x)$ using orthogonal polynomials is largely due to Christoffel in 1877. To introduce these orthogonal polynomials, let us fix the interval of interest to be (a, b). We can define the "scalar product of two functions f and g over a

weight function W" as

$$\langle f|g\rangle \equiv \int_a^b W(x)f(x)g(x)dx \tag{4.5.5}$$

The scalar product is a number, not a function of x. Two functions are said to be *orthogonal* if their scalar product is zero. A function is said to be *normalized* if its scalar product with itself is unity. A set of functions that are all mutually orthogonal and also all individually normalized is called an *orthonormal* set.

We can find a set of polynomials (i) that includes exactly one polynomial of order j, called $p_j(x)$, for each $j = 0, 1, 2, \ldots$, and (ii) all of which are mutually orthogonal over the specified weight function $W(x)$. A constructive procedure for finding such a set is the recurrence relation

$$\begin{aligned} p_{-1}(x) &\equiv 0 \\ p_0(x) &\equiv 1 \\ p_{j+1}(x) &= (x - a_j)p_j(x) - b_j p_{j-1}(x) \qquad j = 0, 1, 2, \ldots \end{aligned} \tag{4.5.6}$$

where

$$\begin{aligned} a_j &= \frac{\langle xp_j|p_j\rangle}{\langle p_j|p_j\rangle} \qquad j = 0, 1, \ldots \\ b_j &= \frac{\langle p_j|p_j\rangle}{\langle p_{j-1}|p_{j-1}\rangle} \qquad j = 1, 2, \ldots \end{aligned} \tag{4.5.7}$$

The coefficient b_0 is arbitrary; we can take it to be zero.

The polynomials defined by (4.5.6) are *monic*, i.e., the coefficient of their leading term [x^j for $p_j(x)$] is unity. If we divide each $p_j(x)$ by the constant $[\langle p_j|p_j\rangle]^{1/2}$ we can render the set of polynomials orthonormal. One also encounters orthogonal polynomials with various other normalizations. You can convert from a given normalization to monic polynomials if you know that the coefficient of x^j in p_j is λ_j, say; then the monic polynomials are obtained by dividing each p_j by λ_j. Note that the coefficients in the recurrence relation (4.5.6) depend on the adopted normalization.

The polynomial $p_j(x)$ can be shown to have exactly j distinct roots in the interval (a, b). Moreover, it can be shown that the roots of $p_j(x)$ "interleave" the $j - 1$ roots of $p_{j-1}(x)$, i.e., there is exactly one root of the former in between each two adjacent roots of the latter. This fact comes in handy if you need to find all the roots: You can start with the one root of $p_1(x)$ and then, in turn, bracket the roots of each higher j, pinning them down at each stage more precisely by Newton's rule or some other root-finding scheme (see Chapter 9).

Why would you ever want to find all the roots of an orthogonal polynomial $p_j(x)$? Because the abscissas of the N-point Gaussian quadrature formulas (4.5.1) and (4.5.4) with weighting function $W(x)$ in the interval (a, b) are precisely the roots of the orthogonal polynomial $p_N(x)$ for the same interval and weighting function. This is the fundamental theorem of Gaussian quadratures, and lets you find the abscissas for any particular case.

Once you know the abscissas $x_1, \ldots, x_N$, you need to find the weights w_j, $j = 1, \ldots, N$. One way to do this (not the most efficient) is to solve the set of linear equations

$$\begin{bmatrix} p_0(x_1) & \cdots & p_0(x_N) \\ p_1(x_1) & \cdots & p_1(x_N) \\ \vdots & & \vdots \\ p_{N-1}(x_1) & \cdots & p_{N-1}(x_N) \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} = \begin{bmatrix} \int_a^b W(x)p_0(x)dx \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{4.5.8}$$

Equation (4.5.8) simply solves for those weights such that the quadrature (4.5.1) gives the correct answer for the integral of the first N orthogonal polynomials. Note that the zeros on the right-hand side of (4.5.8) appear because $p_1(x), \ldots, p_{N-1}(x)$ are all orthogonal to $p_0(x)$, which is a constant. It can be shown that, with those weights, the integral of the *next* $N-1$ polynomials is also exact, so that the quadrature is exact for all polynomials of degree $2N-1$ or less. Another way to evaluate the weights (though one whose proof is beyond our scope) is by the formula

$$w_j = \frac{\langle p_{N-1} | p_{N-1} \rangle}{p_{N-1}(x_j)p'_N(x_j)} \tag{4.5.9}$$

where $p'_N(x_j)$ is the derivative of the orthogonal polynomial at its zero x_j.

The computation of Gaussian quadrature rules thus involves two distinct phases: (i) the generation of the orthogonal polynomials $p_0, \ldots, p_N$, i.e., the computation of the coefficients a_j, b_j in (4.5.6); (ii) the determination of the zeros of $p_N(x)$, and the computation of the associated weights. For the case of the "classical" orthogonal polynomials, the coefficients a_j and b_j are explicitly known (equations 4.5.10 – 4.5.14 below) and phase (i) can be omitted. However, if you are confronted with a "nonclassical" weight function $W(x)$, and you don't know the coefficients a_j and b_j, the construction of the associated set of orthogonal polynomials is not trivial. We discuss it at the end of this section.

Computation of the Abscissas and Weights

This task can range from easy to difficult, depending on how much you already know about your weight function and its associated polynomials. In the case of classical, well-studied, orthogonal polynomials, practically everything is known, including good approximations for their zeros. These can be used as starting guesses, enabling Newton's method (to be discussed in §9.4) to converge very rapidly. Newton's method requires the derivative $p'_N(x)$, which is evaluated by standard relations in terms of p_N and p_{N-1}. The weights are then conveniently evaluated by equation (4.5.9). For the following named cases, this direct root-finding is faster, by a factor of 3 to 5, than any other method.

Here are the weight functions, intervals, and recurrence relations that generate the most commonly used orthogonal polynomials and their corresponding Gaussian quadrature formulas.

Gauss-Legendre:

$$W(x) = 1 \qquad -1 < x < 1$$

$$(j+1)P_{j+1} = (2j+1)xP_j - jP_{j-1} \tag{4.5.10}$$

Gauss-Chebyshev:

$$W(x) = (1-x^2)^{-1/2} \qquad -1 < x < 1$$

$$T_{j+1} = 2xT_j - T_{j-1} \tag{4.5.11}$$

Gauss-Laguerre:

$$W(x) = x^\alpha e^{-x} \qquad 0 < x < \infty$$

$$(j+1)L_{j+1}^\alpha = (-x+2j+\alpha+1)L_j^\alpha - (j+\alpha)L_{j-1}^\alpha \tag{4.5.12}$$

Gauss-Hermite:

$$W(x) = e^{-x^2} \qquad -\infty < x < \infty$$

$$H_{j+1} = 2xH_j - 2jH_{j-1} \tag{4.5.13}$$

Gauss-Jacobi:

$$W(x) = (1-x)^\alpha (1+x)^\beta \qquad -1 < x < 1$$

$$c_j P_{j+1}^{(\alpha,\beta)} = (d_j + e_j x)P_j^{(\alpha,\beta)} - f_j P_{j-1}^{(\alpha,\beta)} \tag{4.5.14}$$

where the coefficients c_j, d_j, e_j, and f_j are given by

$$\begin{aligned} c_j &= 2(j+1)(j+\alpha+\beta+1)(2j+\alpha+\beta) \\ d_j &= (2j+\alpha+\beta+1)(\alpha^2-\beta^2) \\ e_j &= (2j+\alpha+\beta)(2j+\alpha+\beta+1)(2j+\alpha+\beta+2) \\ f_j &= 2(j+\alpha)(j+\beta)(2j+\alpha+\beta+2) \end{aligned} \tag{4.5.15}$$

We now give individual routines that calculate the abscissas and weights for these cases. First comes the most common set of abscissas and weights, those of Gauss-Legendre. The routine, due to G.B. Rybicki, uses equation (4.5.9) in the special form for the Gauss-Legendre case.

$$w_j = \frac{2}{(1-x_j^2)[P_N'(x_j)]^2} \tag{4.5.16}$$

The routine also scales the range of integration from (x_1, x_2) to $(-1, 1)$, and provides abscissas x_j and weights w_j for the Gaussian formula

$$\int_{x_1}^{x_2} f(x)dx = \sum_{j=1}^{N} w_j f(x_j) \tag{4.5.17}$$

```
#include <math.h>
#define EPS 3.0e-11                          EPS is the relative precision.

void gauleg(float x1, float x2, float x[], float w[], int n)
Given the lower and upper limits of integration x1 and x2, and given n, this routine returns
arrays x[1..n] and w[1..n] of length n, containing the abscissas and weights of the Gauss-
Legendre n-point quadrature formula.
{
    int m,j,i;
    double z1,z,xm,xl,pp,p3,p2,p1;           High precision is a good idea for this rou-
                                                 tine.
    m=(n+1)/2;                               The roots are symmetric in the interval, so
    xm=0.5*(x2+x1);                              we only have to find half of them.
    xl=0.5*(x2-x1);
    for (i=1;i<=m;i++) {                     Loop over the desired roots.
        z=cos(3.141592654*(i-0.25)/(n+0.5));
        Starting with the above approximation to the ith root, we enter the main loop of
        refinement by Newton's method.
        do {
            p1=1.0;
            p2=0.0;
            for (j=1;j<=n;j++) {             Loop up the recurrence relation to get the
                p3=p2;                           Legendre polynomial evaluated at z.
                p2=p1;
                p1=((2.0*j-1.0)*z*p2-(j-1.0)*p3)/j;
            }
            p1 is now the desired Legendre polynomial. We next compute pp, its derivative,
            by a standard relation involving also p2, the polynomial of one lower order.
            pp=n*(z*p1-p2)/(z*z-1.0);
            z1=z;
            z=z1-p1/pp;                      Newton's method.
        } while (fabs(z-z1) > EPS);
        x[i]=xm-xl*z;                        Scale the root to the desired interval,
        x[n+1-i]=xm+xl*z;                    and put in its symmetric counterpart.
        w[i]=2.0*xl/((1.0-z*z)*pp*pp);       Compute the weight
        w[n+1-i]=w[i];                       and its symmetric counterpart.
    }
}
```

Next we give three routines that use initial approximations for the roots given by Stroud and Secrest [2]. The first is for Gauss-Laguerre abscissas and weights, to be used with the integration formula

$$\int_0^\infty x^\alpha e^{-x} f(x)dx = \sum_{j=1}^{N} w_j f(x_j) \tag{4.5.18}$$

```
#include <math.h>
#define EPS 3.0e-14                          Increase EPS if you don't have this preci-
#define MAXIT 10                                 sion.

void gaulag(float x[], float w[], int n, float alf)
Given alf, the parameter α of the Laguerre polynomials, this routine returns arrays x[1..n]
and w[1..n] containing the abscissas and weights of the n-point Gauss-Laguerre quadrature
formula. The smallest abscissa is returned in x[1], the largest in x[n].
{
    float gammln(float xx);
    void nrerror(char error_text[]);
    int i,its,j;
    float ai;
```

```
    double p1,p2,p3,pp,z,z1;                    High precision is a good idea for this rou-
                                                    tine.
    for (i=1;i<=n;i++) {                        Loop over the desired roots.
        if (i == 1) {                           Initial guess for the smallest root.
            z=(1.0+alf)*(3.0+0.92*alf)/(1.0+2.4*n+1.8*alf);
        } else if (i == 2) {                    Initial guess for the second root.
            z += (15.0+6.25*alf)/(1.0+0.9*alf+2.5*n);
        } else {                                Initial guess for the other roots.
            ai=i-2;
            z += ((1.0+2.55*ai)/(1.9*ai)+1.26*ai*alf/
                (1.0+3.5*ai))*(z-x[i-2])/(1.0+0.3*alf);
        }
        for (its=1;its<=MAXIT;its++) {          Refinement by Newton's method.
            p1=1.0;
            p2=0.0;
            for (j=1;j<=n;j++) {                Loop up the recurrence relation to get the
                p3=p2;                              Laguerre polynomial evaluated at z.
                p2=p1;
                p1=((2*j-1+alf-z)*p2-(j-1+alf)*p3)/j;
            }
            p1 is now the desired Laguerre polynomial. We next compute pp, its derivative,
            by a standard relation involving also p2, the polynomial of one lower order.
            pp=(n*p1-(n+alf)*p2)/z;
            z1=z;
            z=z1-p1/pp;                         Newton's formula.
            if (fabs(z-z1) <= EPS) break;
        }
        if (its > MAXIT) nrerror("too many iterations in gaulag");
        x[i]=z;                                 Store the root and the weight.
        w[i] = -exp(gammln(alf+n)-gammln((float)n))/(pp*n*p2);
    }
}
```

Next is a routine for Gauss-Hermite abscissas and weights. If we use the "standard" normalization of these functions, as given in equation (4.5.13), we find that the computations overflow for large N because of various factorials that occur. We can avoid this by using instead the orthonormal set of polynomials $\tilde{H}_j$. They are generated by the recurrence

$$\tilde{H}_{-1} = 0, \quad \tilde{H}_0 = \frac{1}{\pi^{1/4}}, \quad \tilde{H}_{j+1} = x\sqrt{\frac{2}{j+1}}\tilde{H}_j - \sqrt{\frac{j}{j+1}}\tilde{H}_{j-1} \tag{4.5.19}$$

The formula for the weights becomes

$$w_j = \frac{2}{[\tilde{H}'_N(x_j)]^2} \tag{4.5.20}$$

while the formula for the derivative with this normalization is

$$\tilde{H}'_j = \sqrt{2j}\tilde{H}_{j-1} \tag{4.5.21}$$

The abscissas and weights returned by gauher are used with the integration formula

$$\int_{-\infty}^{\infty} e^{-x^2} f(x)dx = \sum_{j=1}^{N} w_j f(x_j) \tag{4.5.22}$$

```
#include <math.h>
#define EPS 3.0e-14                          Relative precision.
#define PIM4 0.7511255444649425              1/π^(1/4).
#define MAXIT 10                             Maximum iterations.

void gauher(float x[], float w[], int n)
Given n, this routine returns arrays x[1..n] and w[1..n] containing the abscissas and weights
of the n-point Gauss-Hermite quadrature formula. The largest abscissa is returned in x[1], the
most negative in x[n].
{
    void nrerror(char error_text[]);
    int i,its,j,m;
    double p1,p2,p3,pp,z,z1;                 High precision is a good idea for this rou-
                                                 tine.
    m=(n+1)/2;
    The roots are symmetric about the origin, so we have to find only half of them.
    for (i=1;i<=m;i++) {                     Loop over the desired roots.
        if (i == 1) {                        Initial guess for the largest root.
            z=sqrt((double)(2*n+1))-1.85575*pow((double)(2*n+1),-0.16667);
        } else if (i == 2) {                 Initial guess for the second largest root.
            z -= 1.14*pow((double)n,0.426)/z;
        } else if (i == 3) {                 Initial guess for the third largest root.
            z=1.86*z-0.86*x[1];
        } else if (i == 4) {                 Initial guess for the fourth largest root.
            z=1.91*z-0.91*x[2];
        } else {                             Initial guess for the other roots.
            z=2.0*z-x[i-2];
        }
        for (its=1;its<=MAXIT;its++) {       Refinement by Newton's method.
            p1=PIM4;
            p2=0.0;
            for (j=1;j<=n;j++) {             Loop up the recurrence relation to get
                p3=p2;                           the Hermite polynomial evaluated at
                p2=p1;                           z.
                p1=z*sqrt(2.0/j)*p2-sqrt(((double)(j-1))/j)*p3;
            }
            p1 is now the desired Hermite polynomial. We next compute pp, its derivative, by
            the relation (4.5.21) using p2, the polynomial of one lower order.
            pp=sqrt((double)2*n)*p2;
            z1=z;
            z=z1-p1/pp;                      Newton's formula.
            if (fabs(z-z1) <= EPS) break;
        }
        if (its > MAXIT) nrerror("too many iterations in gauher");
        x[i]=z;                              Store the root
        x[n+1-i] = -z;                       and its symmetric counterpart.
        w[i]=2.0/(pp*pp);                    Compute the weight
        w[n+1-i]=w[i];                       and its symmetric counterpart.
    }
}
```

Finally, here is a routine for Gauss-Jacobi abscissas and weights, which implement the integration formula

$$\int_{-1}^{1}(1-x)^{\alpha}(1+x)^{\beta}f(x)dx=\sum_{j=1}^{N}w_j f(x_j) \tag{4.5.23}$$

```
#include <math.h>
#define EPS 3.0e-14                          Increase EPS if you don't have this preci-
#define MAXIT 10                                sion.

void gaujac(float x[], float w[], int n, float alf, float bet)
Given alf and bet, the parameters α and β of the Jacobi polynomials, this routine returns
arrays x[1..n] and w[1..n] containing the abscissas and weights of the n-point Gauss-Jacobi
quadrature formula. The largest abscissa is returned in x[1], the smallest in x[n].
{
    float gammln(float xx);
    void nrerror(char error_text[]);
    int i,its,j;
    float alfbet,an,bn,r1,r2,r3;
    double a,b,c,p1,p2,p3,pp,temp,z,z1;      High precision is a good idea for this rou-
                                                tine.
    for (i=1;i<=n;i++) {                     Loop over the desired roots.
        if (i == 1) {                        Initial guess for the largest root.
            an=alf/n;
            bn=bet/n;
            r1=(1.0+alf)*(2.78/(4.0+n*n)+0.768*an/n);
            r2=1.0+1.48*an+0.96*bn+0.452*an*an+0.83*an*bn;
            z=1.0-r1/r2;
        } else if (i == 2) {                 Initial guess for the second largest root.
            r1=(4.1+alf)/((1.0+alf)*(1.0+0.156*alf));
            r2=1.0+0.06*(n-8.0)*(1.0+0.12*alf)/n;
            r3=1.0+0.012*bet*(1.0+0.25*fabs(alf))/n;
            z -= (1.0-z)*r1*r2*r3;
        } else if (i == 3) {                 Initial guess for the third largest root.
            r1=(1.67+0.28*alf)/(1.0+0.37*alf);
            r2=1.0+0.22*(n-8.0)/n;
            r3=1.0+8.0*bet/((6.28+bet)*n*n);
            z -= (x[1]-z)*r1*r2*r3;
        } else if (i == n-1) {               Initial guess for the second smallest root.
            r1=(1.0+0.235*bet)/(0.766+0.119*bet);
            r2=1.0/(1.0+0.639*(n-4.0)/(1.0+0.71*(n-4.0)));
            r3=1.0/(1.0+20.0*alf/((7.5+alf)*n*n));
            z += (z-x[n-3])*r1*r2*r3;
        } else if (i == n) {                 Initial guess for the smallest root.
            r1=(1.0+0.37*bet)/(1.67+0.28*bet);
            r2=1.0/(1.0+0.22*(n-8.0)/n);
            r3=1.0/(1.0+8.0*alf/((6.28+alf)*n*n));
            z += (z-x[n-2])*r1*r2*r3;
        } else {                             Initial guess for the other roots.
            z=3.0*x[i-1]-3.0*x[i-2]+x[i-3];
        }
        alfbet=alf+bet;
        for (its=1;its<=MAXIT;its++) {       Refinement by Newton's method.
            temp=2.0+alfbet;                 Start the recurrence with P_0 and P_1 to avoid
            p1=(alf-bet+temp*z)/2.0;            a division by zero when α + β = 0 or
            p2=1.0;                             −1.
            for (j=2;j<=n;j++) {             Loop up the recurrence relation to get the
                p3=p2;                          Jacobi polynomial evaluated at z.
                p2=p1;
                temp=2*j+alfbet;
                a=2*j*(j+alfbet)*(temp-2.0);
                b=(temp-1.0)*(alf*alf-bet*bet+temp*(temp-2.0)*z);
                c=2.0*(j-1+alf)*(j-1+bet)*temp;
                p1=(b*p2-c*p3)/a;
            }
            pp=(n*(alf-bet-temp*z)*p1+2.0*(n+alf)*(n+bet)*p2)/(temp*(1.0-z*z));
            p1 is now the desired Jacobi polynomial. We next compute pp, its derivative, by
            a standard relation involving also p2, the polynomial of one lower order.
            z1=z;
            z=z1-p1/pp;                      Newton's formula.
```

```
            if (fabs(z-z1) <= EPS) break;
        }
        if (its > MAXIT) nrerror("too many iterations in gaujac");
        x[i]=z;                                Store the root and the weight.
        w[i]=exp(gammln(alf+n)+gammln(bet+n)-gammln(n+1.0)-
            gammln(n+alfbet+1.0))*temp*pow(2.0,alfbet)/(pp*p2);
    }
}
```

Legendre polynomials are special cases of Jacobi polynomials with $\alpha = \beta = 0$, but it is worth having the separate routine for them, `gauleg`, given above. Chebyshev polynomials correspond to $\alpha = \beta = -1/2$ (see §5.8). They have analytic abscissas and weights:

$$x_j = \cos\left(\frac{\pi(j-\frac{1}{2})}{N}\right)$$
$$w_j = \frac{\pi}{N} \tag{4.5.24}$$

Case of Known Recurrences

Turn now to the case where you do not know good initial guesses for the zeros of your orthogonal polynomials, but you do have available the coefficients a_j and b_j that generate them. As we have seen, the zeros of $p_N(x)$ are the abscissas for the N-point Gaussian quadrature formula. The most useful computational formula for the weights is equation (4.5.9) above, since the derivative p'_N can be efficiently computed by the derivative of (4.5.6) in the general case, or by special relations for the classical polynomials. Note that (4.5.9) is valid as written only for monic polynomials; for other normalizations, there is an extra factor of λ_N/λ_{N-1}, where λ_N is the coefficient of x^N in p_N.

Except in those special cases already discussed, the best way to find the abscissas is *not* to use a root-finding method like Newton's method on $p_N(x)$. Rather, it is generally faster to use the Golub-Welsch [3] algorithm, which is based on a result of Wilf [4]. This algorithm notes that if you bring the term xp_j to the left-hand side of (4.5.6) and the term p_{j+1} to the right-hand side, the recurrence relation can be written in matrix form as

$$x\begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{N-2} \\ p_{N-1} \end{bmatrix} = \begin{bmatrix} a_0 & 1 & & & \\ b_1 & a_1 & 1 & & \\ & \vdots & \vdots & & \\ & & b_{N-2} & a_{N-2} & 1 \\ & & & b_{N-1} & a_{N-1} \end{bmatrix} \cdot \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{N-2} \\ p_{N-1} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ p_N \end{bmatrix}$$

or

$$x\mathbf{p} = \mathbf{T}\cdot\mathbf{p} + p_N\mathbf{e}_{N-1} \tag{4.5.25}$$

Here $\mathbf{T}$ is a tridiagonal matrix, $\mathbf{p}$ is a column vector of $p_0, p_1, \ldots, p_{N-1}$, and $\mathbf{e}_{N-1}$ is a unit vector with a 1 in the $(N-1)$st (last) position and zeros elsewhere. The matrix $\mathbf{T}$ can be symmetrized by a diagonal similarity transformation $\mathbf{D}$ to give

$$\mathbf{J} = \mathbf{DTD}^{-1} = \begin{bmatrix} a_0 & \sqrt{b_1} & & & \\ \sqrt{b_1} & a_1 & \sqrt{b_2} & & \\ & \vdots & \vdots & & \\ & & \sqrt{b_{N-2}} & a_{N-2} & \sqrt{b_{N-1}} \\ & & & \sqrt{b_{N-1}} & a_{N-1} \end{bmatrix} \tag{4.5.26}$$

The matrix $\mathbf{J}$ is called the *Jacobi matrix* (not to be confused with other matrices named after Jacobi that arise in completely different problems!). Now we see from (4.5.25) that

$p_N(x_j) = 0$ is equivalent to x_j being an eigenvalue of $\mathbf{T}$. Since eigenvalues are preserved by a similarity transformation, x_j is an eigenvalue of the symmetric tridiagonal matrix $\mathbf{J}$. Moreover, Wilf [4] shows that if $\mathbf{v}_j$ is the eigenvector corresponding to the eigenvalue x_j, normalized so that $\mathbf{v} \cdot \mathbf{v} = 1$, then

$$w_j = \mu_0 v_{j,1}^2 \tag{4.5.27}$$

where

$$\mu_0 = \int_a^b W(x)\,dx \tag{4.5.28}$$

and where $v_{j,1}$ is the first component of $\mathbf{v}$. As we shall see in Chapter 11, finding all eigenvalues and eigenvectors of a symmetric tridiagonal matrix is a relatively efficient and well-conditioned procedure. We accordingly give a routine, `gaucof`, for finding the abscissas and weights, given the coefficients a_j and b_j. Remember that if you know the recurrence relation for orthogonal polynomials that are not normalized to be monic, you can easily convert it to monic form by means of the quantities λ_j.

```
#include <math.h>
#include "nrutil.h"

void gaucof(int n, float a[], float b[], float amu0, float x[], float w[])
Computes the abscissas and weights for a Gaussian quadrature formula from the Jacobi matrix.
On input, a[1..n] and b[1..n] are the coefficients of the recurrence relation for the set of
monic orthogonal polynomials. The quantity mu0 ≡ ∫_a^b W(x) dx is input as amu0. The abscissas
x[1..n] are returned in descending order, with the corresponding weights in w[1..n]. The
arrays a and b are modified. Execution can be speeded up by modifying tqli and eigsrt to
compute only the first component of each eigenvector.
{
    void eigsrt(float d[], float **v, int n);
    void tqli(float d[], float e[], int n, float **z);
    int i,j;
    float **z;

    z=matrix(1,n,1,n);
    for (i=1;i<=n;i++) {
        if (i != 1) b[i]=sqrt(b[i]);        Set up superdiagonal of Jacobi matrix.
        for (j=1;j<=n;j++) z[i][j]=(float)(i == j);
        Set up identity matrix for tqli to compute eigenvectors.
    }
    tqli(a,b,n,z);
    eigsrt(a,z,n);                          Sort eigenvalues into descending order.
    for (i=1;i<=n;i++) {
        x[i]=a[i];
        w[i]=amu0*z[1][i]*z[1][i];          Equation (4.5.27).
    }
    free_matrix(z,1,n,1,n);
}
```

Orthogonal Polynomials with Nonclassical Weights

This somewhat specialized subsection will tell you what to do if your weight function is not one of the classical ones dealt with above and you do not know the a_j's and b_j's of the recurrence relation (4.5.6) to use in `gaucof`. Then, a method of finding the a_j's and b_j's is needed.

The *procedure of Stieltjes* is to compute a_0 from (4.5.7), then $p_1(x)$ from (4.5.6). Knowing p_0 and p_1, we can compute a_1 and b_1 from (4.5.7), and so on. But how are we to compute the inner products in (4.5.7)?

The textbook approach is to represent each $p_j(x)$ explicitly as a polynomial in x and to compute the inner products by multiplying out term by term. This will be feasible if we know the first $2N$ moments of the weight function,

$$\mu_j = \int_a^b x^j W(x)dx \qquad j = 0, 1, \ldots, 2N-1 \tag{4.5.29}$$

However, the solution of the resulting set of algebraic equations for the coefficients a_j and b_j in terms of the moments μ_j is in general *extremely* ill-conditioned. Even in double precision, it is not unusual to lose all accuracy by the time $N = 12$. We thus reject any procedure based on the moments (4.5.29).

Sack and Donovan [5] discovered that the numerical stability is greatly improved if, instead of using powers of x as a set of basis functions to represent the p_j's, one uses some other known set of orthogonal polynomials $\pi_j(x)$, say. Roughly speaking, the improved stability occurs because the polynomial basis "samples" the interval (a, b) better than the power basis when the inner product integrals are evaluated, especially if its weight function resembles $W(x)$.

So assume that we know the *modified moments*

$$\nu_j = \int_a^b \pi_j(x)W(x)dx \qquad j = 0, 1, \ldots, 2N-1 \tag{4.5.30}$$

where the π_j's satisfy a recurrence relation analogous to (4.5.6),

$$\begin{aligned} \pi_{-1}(x) &\equiv 0 \\ \pi_0(x) &\equiv 1 \\ \pi_{j+1}(x) &= (x - \alpha_j)\pi_j(x) - \beta_j\pi_{j-1}(x) \qquad j = 0, 1, 2, \ldots \end{aligned} \tag{4.5.31}$$

and the coefficients α_j, β_j are known explicitly. Then Wheeler [6] has given an efficient $O(N^2)$ algorithm equivalent to that of Sack and Donovan for finding a_j and b_j via a set of intermediate quantities

$$\sigma_{k,l} = \langle p_k | \pi_l \rangle \qquad k, l \geq -1 \tag{4.5.32}$$

Initialize

$$\begin{aligned} \sigma_{-1,l} &= 0 && l = 1, 2, \ldots, 2N-2 \\ \sigma_{0,l} &= \nu_l && l = 0, 1, \ldots, 2N-1 \\ a_0 &= \alpha_0 + \frac{\nu_1}{\nu_0} \\ b_0 &= 0 \end{aligned} \tag{4.5.33}$$

Then, for $k = 1, 2, \ldots, N-1$, compute

$$\begin{aligned} \sigma_{k,l} &= \sigma_{k-1,l+1} - (a_{k-1} - \alpha_l)\sigma_{k-1,l} - b_{k-1}\sigma_{k-2,l} + \beta_l\sigma_{k-1,l-1} \\ &\qquad\qquad l = k, k+1, \ldots, 2N-k-1 \\ a_k &= \alpha_k - \frac{\sigma_{k-1,k}}{\sigma_{k-1,k-1}} + \frac{\sigma_{k,k+1}}{\sigma_{k,k}} \\ b_k &= \frac{\sigma_{k,k}}{\sigma_{k-1,k-1}} \end{aligned} \tag{4.5.34}$$

Note that the normalization factors can also easily be computed if needed:

$$\begin{aligned} \langle p_0 | p_0 \rangle &= \nu_0 \\ \langle p_j | p_j \rangle &= b_j \langle p_{j-1} | p_{j-1} \rangle \qquad j = 1, 2, \ldots \end{aligned} \tag{4.5.35}$$

You can find a derivation of the above algorithm in Ref. [7].

Wheeler's algorithm requires that the modified moments (4.5.30) be accurately computed. In practical cases there is often a closed form, or else recurrence relations can be used. The

algorithm is extremely successful for *finite* intervals (a, b). For infinite intervals, the algorithm does not completely remove the ill-conditioning. In this case, Gautschi [8,9] recommends reducing the interval to a finite interval by a change of variable, and then using a suitable discretization procedure to compute the inner products. You will have to consult the references for details.

We give the routine `orthog` for generating the coefficients a_j and b_j by Wheeler's algorithm, given the coefficients α_j and β_j, and the modified moments ν_j. For consistency with `gaucof`, the vectors α, β, a and b are 1-based. Correspondingly, we increase the indices of the σ matrix by 2, i.e., `sig[k,l]` $= \sigma_{k-2,l-2}$.

```
#include "nrutil.h"

void orthog(int n, float anu[], float alpha[], float beta[], float a[],
    float b[])
```

Computes the coefficients a_j and b_j, $j = 0, \ldots N-1$, of the recurrence relation for monic orthogonal polynomials with weight function $W(x)$ by Wheeler's algorithm. On input, the arrays `alpha[1..2*n-1]` and `beta[1..2*n-1]` are the coefficients α_j and β_j, $j = 0, \ldots 2N-2$, of the recurrence relation for the chosen basis of orthogonal polynomials. The modified moments ν_j are input in `anu[1..2*n]`. The first `n` coefficients are returned in `a[1..n]` and `b[1..n]`.

```
{
    int k,l;
    float **sig;
    int looptmp;

    sig=matrix(1,2*n+1,1,2*n+1);
    looptmp=2*n;
    for (l=3;l<=looptmp;l++) sig[1][l]=0.0;          Initialization, Equation (4.5.33).
    looptmp++;
    for (l=2;l<=looptmp;l++) sig[2][l]=anu[l-1];
    a[1]=alpha[1]+anu[2]/anu[1];
    b[1]=0.0;
    for (k=3;k<=n+1;k++) {                           Equation (4.5.34).
        looptmp=2*n-k+3;
        for (l=k;l<=looptmp;l++) {
            sig[k][l]=sig[k-1][l+1]+(alpha[l-1]-a[k-2])*sig[k-1][l]-
                b[k-2]*sig[k-2][l]+beta[l-1]*sig[k-1][l-1];
        }
        a[k-1]=alpha[k-1]+sig[k][k+1]/sig[k][k]-sig[k-1][k]/sig[k-1][k-1];
        b[k-1]=sig[k][k]/sig[k-1][k-1];
    }
    free_matrix(sig,1,2*n+1,1,2*n+1);
}
```

As an example of the use of `orthog`, consider the problem [7] of generating orthogonal polynomials with the weight function $W(x) = -\log x$ on the interval $(0, 1)$. A suitable set of π_j's is the shifted Legendre polynomials

$$\pi_j = \frac{(j!)^2}{(2j)!} P_j(2x-1) \tag{4.5.36}$$

The factor in front of P_j makes the polynomials monic. The coefficients in the recurrence relation (4.5.31) are

$$\begin{aligned} \alpha_j &= \frac{1}{2} && j = 0, 1, \ldots \\ \beta_j &= \frac{1}{4(4-j^{-2})} && j = 1, 2, \ldots \end{aligned} \tag{4.5.37}$$

while the modified moments are

$$\nu_j = \begin{cases} 1 & j = 0 \\ \dfrac{(-1)^j (j!)^2}{j(j+1)(2j)!} & j \geq 1 \end{cases} \tag{4.5.38}$$

A call to `orthog` with this input allows one to generate the required polynomials to machine accuracy for very large N, and hence do Gaussian quadrature with this weight function. Before Sack and Donovan's observation, this seemingly simple problem was essentially intractable.

Extensions of Gaussian Quadrature

There are many different ways in which the ideas of Gaussian quadrature have been extended. One important extension is the case of *preassigned nodes*: Some points are required to be included in the set of abscissas, and the problem is to choose the weights and the remaining abscissas to maximize the degree of exactness of the the quadrature rule. The most common cases are *Gauss-Radau* quadrature, where one of the nodes is an endpoint of the interval, either a or b, and *Gauss-Lobatto* quadrature, where both a and b are nodes. Golub [10] has given an algorithm similar to `gaucof` for these cases.

The second important extension is the *Gauss-Kronrod* formulas. For ordinary Gaussian quadrature formulas, as N increases the sets of abscissas have no points in common. This means that if you compare results with increasing N as a way of estimating the quadrature error, you cannot reuse the previous function evaluations. Kronrod [11] posed the problem of searching for optimal sequences of rules, each of which reuses all abscissas of its predecessor. If one starts with $N = m$, say, and then adds n new points, one has $2n + m$ free parameters: the n new abscissas and weights, and m new weights for the fixed previous abscissas. The maximum degree of exactness one would expect to achieve would therefore be $2n + m - 1$. The question is whether this maximum degree of exactness can actually be achieved in practice, when the abscissas are required to all lie inside (a, b). The answer to this question is not known in general.

Kronrod showed that if you choose $n = m + 1$, an optimal extension can be found for Gauss-Legendre quadrature. Patterson [12] showed how to compute continued extensions of this kind. Sequences such as $N = 10, 21, 43, 87, \ldots$ are popular in automatic quadrature routines [13] that attempt to integrate a function until some specified accuracy has been achieved.

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), §25.4. [1]

Stroud, A.H., and Secrest, D. 1966, *Gaussian Quadrature Formulas* (Englewood Cliffs, NJ: Prentice-Hall). [2]

Golub, G.H., and Welsch, J.H. 1969, *Mathematics of Computation*, vol. 23, pp. 221–230 and A1–A10. [3]

Wilf, H.S. 1962, *Mathematics for the Physical Sciences* (New York: Wiley), Problem 9, p. 80. [4]

Sack, R.A., and Donovan, A.F. 1971/72, *Numerische Mathematik*, vol. 18, pp. 465–478. [5]

Wheeler, J.C. 1974, *Rocky Mountain Journal of Mathematics*, vol. 4, pp. 287–296. [6]

Gautschi, W. 1978, in *Recent Advances in Numerical Analysis*, C. de Boor and G.H. Golub, eds. (New York: Academic Press), pp. 45–72. [7]

Gautschi, W. 1981, in *E.B. Christoffel*, P.L. Butzer and F. Fehér, eds. (Basel: Birkhauser Verlag), pp. 72–147. [8]

Gautschi, W. 1990, in *Orthogonal Polynomials*, P. Nevai, ed. (Dordrecht: Kluwer Academic Publishers), pp. 181–216. [9]

Golub, G.H. 1973, *SIAM Review*, vol. 15, pp. 318–334. [10]

Kronrod, A.S. 1964, *Doklady Akademii Nauk SSSR*, vol. 154, pp. 283–286 (in Russian). [11]

Patterson, T.N.L. 1968, *Mathematics of Computation*, vol. 22, pp. 847–856 and C1–C11; 1969, *op. cit.*, vol. 23, p. 892. [12]

Piessens, R., de Doncker, E., Uberhuber, C.W., and Kahaner, D.K. 1983, *QUADPACK: A Subroutine Package for Automatic Integration* (New York: Springer-Verlag). [13]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §3.6.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §6.5.

Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley), §§2.9–2.10.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §§4.4–4.8.

4.6 Multidimensional Integrals

Integrals of functions of several variables, over regions with dimension greater than one, are *not easy*. There are two reasons for this. First, the number of function evaluations needed to sample an N-dimensional space increases as the Nth power of the number needed to do a one-dimensional integral. If you need 30 function evaluations to do a one-dimensional integral crudely, then you will likely need on the order of 30000 evaluations to reach the same crude level for a three-dimensional integral. Second, the region of integration in N-dimensional space is defined by an $N-1$ dimensional boundary which can itself be terribly complicated: It need not be convex or simply connected, for example. By contrast, the boundary of a one-dimensional integral consists of two numbers, its upper and lower limits.

The first question to be asked, when faced with a multidimensional integral, is, "can it be reduced analytically to a lower dimensionality?" For example, so-called *iterated integrals* of a function of one variable $f(t)$ can be reduced to one-dimensional integrals by the formula

$$\begin{aligned}\int_0^x dt_n \int_0^{t_n} dt_{n-1} \cdots \int_0^{t_3} dt_2 \int_0^{t_2} f(t_1)dt_1 \\ = \frac{1}{(n-1)!}\int_0^x (x-t)^{n-1} f(t)dt\end{aligned} \tag{4.6.1}$$

Alternatively, the function may have some special symmetry in the way it depends on its independent variables. If the boundary also has this symmetry, then the dimension can be reduced. In three dimensions, for example, the integration of a spherically symmetric function over a spherical region reduces, in polar coordinates, to a one-dimensional integral.

The next questions to be asked will guide your choice between two entirely different approaches to doing the problem. The questions are: Is the shape of the boundary of the region of integration simple or complicated? Inside the region, is the integrand smooth and simple, or complicated, or locally strongly peaked? Does

the problem require high accuracy, or does it require an answer accurate only to a percent, or a few percent?

If your answers are that the boundary is complicated, the integrand is *not* strongly peaked in very small regions, and relatively low accuracy is tolerable, then your problem is a good candidate for *Monte Carlo integration*. This method is very straightforward to program, in its cruder forms. One needs only to know a region with simple boundaries that *includes* the complicated region of integration, plus a method of determining whether a random point is inside or outside the region of integration. Monte Carlo integration evaluates the function at a random sample of points, and estimates its integral based on that random sample. We will discuss it in more detail, and with more sophistication, in Chapter 7.

If the boundary is simple, and the function is very smooth, then the remaining approaches, breaking up the problem into repeated one-dimensional integrals, or multidimensional Gaussian quadratures, will be effective and relatively fast [1]. If you require high accuracy, these approaches are in any case the *only* ones available to you, since Monte Carlo methods are by nature asymptotically slow to converge.

For low accuracy, use repeated one-dimensional integration or multidimensional Gaussian quadratures when the integrand is slowly varying and smooth in the region of integration, Monte Carlo when the integrand is oscillatory or discontinuous, but not strongly peaked in small regions.

If the integrand *is* strongly peaked in small regions, and you know where those regions are, break the integral up into several regions so that the integrand is smooth in each, and do each separately. If you don't know where the strongly peaked regions are, you might as well (at the level of sophistication of this book) quit: It is hopeless to expect an integration routine to search out unknown pockets of large contribution in a huge N-dimensional space. (But see §7.8.)

If, on the basis of the above guidelines, you decide to pursue the repeated one-dimensional integration approach, here is how it works. For definiteness, we will consider the case of a three-dimensional integral in x, y, z-space. Two dimensions, or more than three dimensions, are entirely analogous.

The first step is to specify the region of integration by (i) its lower and upper limits in x, which we will denote x_1 and x_2; (ii) its lower and upper limits in y at a specified value of x, denoted $y_1(x)$ and $y_2(x)$; and (iii) its lower and upper limits in z at specified x and y, denoted $z_1(x, y)$ and $z_2(x, y)$. In other words, find the numbers x_1 and x_2, and the functions $y_1(x)$, $y_2(x)$, $z_1(x, y)$, and $z_2(x, y)$ such that

$$\begin{aligned} I &\equiv \int\int\int dx\, dy\, dz\, f(x, y, z) \\ &= \int_{x_1}^{x_2} dx \int_{y_1(x)}^{y_2(x)} dy \int_{z_1(x,y)}^{z_2(x,y)} dz\, f(x, y, z) \end{aligned} \tag{4.6.2}$$

For example, a two-dimensional integral over a circle of radius one centered on the origin becomes

$$\int_{-1}^{1} dx \int_{-\sqrt{1-x^2}}^{\sqrt{1-x^2}} dy\, f(x, y) \tag{4.6.3}$$

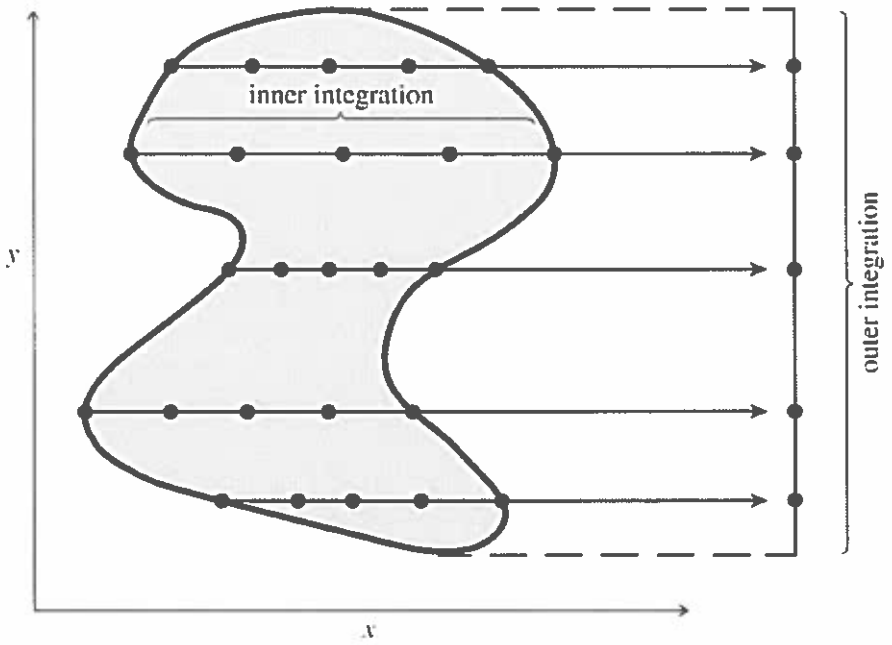

Figure 4.6.1. Function evaluations for a two-dimensional integral over an irregular region, shown schematically. The outer integration routine, in y, requests values of the inner, x, integral at locations along the y axis of its own choosing. The inner integration routine then evaluates the function at x locations suitable to *it*. This is more accurate in general than, e.g., evaluating the function on a Cartesian mesh of points.

Now we can define a function $G(x, y)$ that does the innermost integral,

$$G(x, y) \equiv \int_{z_1(x,y)}^{z_2(x,y)} f(x, y, z)dz \tag{4.6.4}$$

and a function $H(x)$ that does the integral of $G(x, y)$,

$$H(x) \equiv \int_{y_1(x)}^{y_2(x)} G(x, y)dy \tag{4.6.5}$$

and finally our answer as an integral over $H(x)$

$$I = \int_{x_1}^{x_2} H(x)dx \tag{4.6.6}$$

In an implementation of equations (4.6.4)–(4.6.6), some basic one-dimensional integration routine (e.g., `qgaus` in the program following) gets called recursively: once to evaluate the outer integral I, then many times to evaluate the middle integral H, then even more times to evaluate the inner integral G (see Figure 4.6.1). Current values of x and y, and the pointer to your function `func`, are passed "over the head" of the intermediate calls through static top-level variables.

```
static float xsav,ysav;
static float (*nrfunc)(float,float,float);

float quad3d(float (*func)(float, float, float), float x1, float x2)
```

Returns the integral of a user-supplied function `func` over a three-dimensional region specified by the limits `x1`, `x2`, and by the user-supplied functions `yy1`, `yy2`, `z1`, and `z2`, as defined in (4.6.2). (The functions y_1 and y_2 are here called `yy1` and `yy2` to avoid conflict with the names of Bessel functions in some C libraries). Integration is performed by calling `qgaus` recursively.

```
{
    float qgaus(float (*func)(float), float a, float b);
    float f1(float x);

    nrfunc=func;
    return qgaus(f1,x1,x2);
}

float f1(float x)                    This is I1 of eq. (4.6.5).
{
    float qgaus(float (*func)(float), float a, float b);
    float f2(float y);
    float yy1(float),yy2(float);

    xsav=x;
    return qgaus(f2,yy1(x),yy2(x));
}

float f2(float y)                    This is G of eq. (4.6.4).
{
    float qgaus(float (*func)(float), float a, float b);
    float f3(float z);
    float z1(float,float),z2(float,float);

    ysav=y;
    return qgaus(f3,z1(xsav,y),z2(xsav,y));
}

float f3(float z)                    The integrand f(x, y, z) evaluated at fixed x and y.
{
    return (*nrfunc)(xsav,ysav,z);
}
```

The necessary user-supplied functions have the following prototypes:

```
float func(float x,float y,float z);      The 3-dimensional function to be integrated.
float yy1(float x);
float yy2(float x);
float z1(float x,float y);
float z2(float x,float y);
```

CITED REFERENCES AND FURTHER READING:

Stroud, A.H. 1971, *Approximate Calculation of Multiple Integrals* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §7.7, p. 318.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §6.2.5, p. 307.

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), equations 25.4.58ff.