Integration of Functions

CHAPTER 4

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 17 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 17. (But take a look at §4.7 first.)

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 17, 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. A related problem is the evaluation of integrals with long oscillatory tails. This is discussed at the end of §5.3.

Multidimensional integrals are a whole 'nother multidimensional bag of worms. Section 4.8 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*; reprinted 1994 (New York: Dover), Chapter 7.

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

Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), Chapter 3.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), Chapter 4.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 2003 (New York: Dover), §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

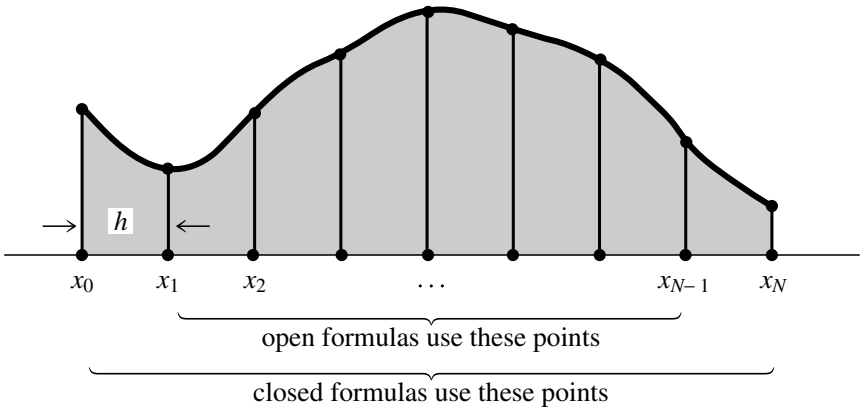

Figure 4.1.1. Quadrature formulas with equally spaced abscissas compute the integral of a function between x_0 and x_N. Closed formulas evaluate the function on the boundary points, while open formulas refrain from doing so (useful if the evaluation algorithm breaks down on the boundary points).

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 midpoint rule," equation 4.1.19; see §4.2), the classical formulas are almost entirely useless. They are museum pieces, but beautiful ones; we now enter the museum. (You can skip to §4.2 if you are not touristically inclined.)

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

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

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.

4.1.1 Closed Newton-Cotes Formulas

Trapezoidal rule:

$$\int_{x_0}^{x_1} f(x)dx = h\left[\frac{1}{2}f_0 + \frac{1}{2}f_1\right] \quad + O(h^3 f'') \tag{4.1.3}$$

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_0 and x_1). 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:

$$\int_{x_0}^{x_2} f(x)dx = h\left[\frac{1}{3}f_0 + \frac{4}{3}f_1 + \frac{1}{3}f_2\right] \quad + O(h^5 f^{(4)}) \tag{4.1.4}$$

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:

$$\int_{x_0}^{x_3} f(x)dx = h\left[\frac{3}{8}f_0 + \frac{9}{8}f_1 + \frac{9}{8}f_2 + \frac{3}{8}f_3\right] \quad + O(h^5 f^{(4)}) \tag{4.1.5}$$

The five-point formula again benefits from a cancellation:

Bode's rule:

$$\int_{x_0}^{x_4} f(x)dx = h\left[\frac{14}{45}f_0 + \frac{64}{45}f_1 + \frac{24}{45}f_2 + \frac{64}{45}f_3 + \frac{14}{45}f_4\right] \quad + O(h^7 f^{(6)}) \tag{4.1.6}$$

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.

4.1.2 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 §4.6.

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 later 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.

4.1.3 Extended Formulas (Closed)

If we use equation (4.1.3) $N - 1$ times to do the integration in the intervals $(x_0, x_1), (x_1, x_2), \ldots, (x_{N-2}, x_{N-1})$ and then add the results, we obtain an "extended" or "composite" formula for the integral from x_0 to x_{N-1}.

Extended trapezoidal rule:

$$\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{1}{2}f_0 + f_1 + f_2 + \cdots + f_{N-2} + \frac{1}{2}f_{N-1}\right] \quad + O\left(\frac{(b-a)^3 f''}{N^2}\right) \tag{4.1.11}$$

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; it is the basis for most practical quadrature schemes.

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

$$\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{5}{12}f_0 + \frac{13}{12}f_1 + f_2 + f_3 + \cdots + f_{N-3} + \frac{13}{12}f_{N-2} + \frac{5}{12}f_{N-1}\right] + O\left(\frac{1}{N^3}\right) \tag{4.1.12}$$

(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:*

$$\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{1}{3}f_0 + \frac{4}{3}f_1 + \frac{2}{3}f_2 + \frac{4}{3}f_3 + \cdots + \frac{2}{3}f_{N-3} + \frac{4}{3}f_{N-2} + \frac{1}{3}f_{N-1}\right] + O\left(\frac{1}{N^4}\right) \tag{4.1.13}$$

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

$$\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{3}{8}f_0 + \frac{7}{6}f_1 + \frac{23}{24}f_2 + f_3 + f_4 + \cdots + f_{N-5} + f_{N-4} + \frac{23}{24}f_{N-3} + \frac{7}{6}f_{N-2} + \frac{3}{8}f_{N-1}\right] + O\left(\frac{1}{N^4}\right) \tag{4.1.14}$$

This equation is constructed by fitting cubic polynomials through successive groups of four points; we defer details to §19.3, 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 steps 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.

4.1.4 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

$$\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{3}{2}f_1 + f_2 + f_3 + \cdots + f_{N-3} + \frac{3}{2}f_{N-2}\right] \quad + O\left(\frac{1}{N^2}\right) \tag{4.1.15}$$

Equations (4.1.8) and (4.1.12) give

$$\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{23}{12}f_1 + \frac{7}{12}f_2 + f_3 + f_4 + \cdots + f_{N-4} + \frac{7}{12}f_{N-3} + \frac{23}{12}f_{N-2}\right] \quad + O\left(\frac{1}{N^3}\right) \tag{4.1.16}$$

Equations (4.1.9) and (4.1.13) give

$$\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{27}{12}f_1 + 0 + \frac{13}{12}f_3 + \frac{4}{3}f_4 + \cdots + \frac{4}{3}f_{N-5} + \frac{13}{12}f_{N-4} + 0 + \frac{27}{12}f_{N-2}\right] \quad + O\left(\frac{1}{N^4}\right) \tag{4.1.17}$$

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

$$\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{55}{24}f_1 - \frac{1}{6}f_2 + \frac{11}{8}f_3 + f_4 + f_5 + f_6 + \cdots + f_{N-6} + f_{N-5} + \frac{11}{8}f_{N-4} - \frac{1}{6}f_{N-3} + \frac{55}{24}f_{N-2}\right] + O\left(\frac{1}{N^4}\right) \tag{4.1.18}$$

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):

$$\int_{x_0}^{x_{N-1}} f(x)dx = h[f_{1/2} + f_{3/2} + f_{5/2} + \cdots + f_{N-5/2} + f_{N-3/2}] \quad + O\left(\frac{1}{N^2}\right) \tag{4.1.19}$$

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

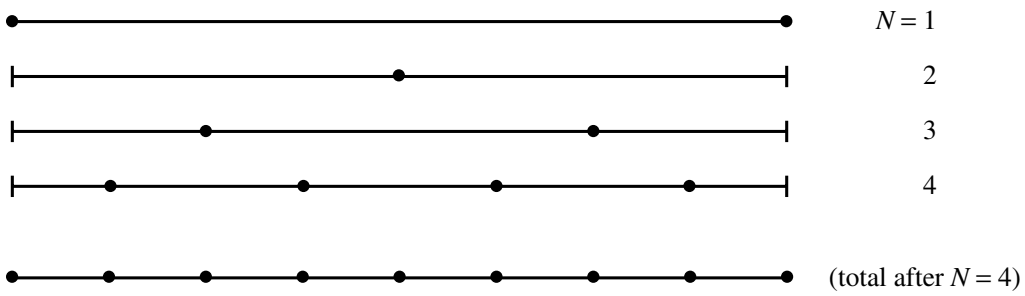

Figure 4.2.1. Sequential calls to the routine `Trapzd` incorporate the information from previous calls and evaluate the integrand only at those new points necessary to refine the grid. The bottom line shows the totality of function evaluations after the fourth call. The routine `qsimp`, by weighting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

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:

$$\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{23}{12}f_1 + \frac{7}{12}f_2 + f_3 + f_4 + \cdots + f_{N-3} + \frac{13}{12}f_{N-2} + \frac{5}{12}f_{N-1}\right] + O\left(\frac{1}{N^3}\right) \tag{4.1.20}$$

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at `http://www.nr.com/aands`, §25.4.[1]

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods*; reprinted 1994 (New York: Dover), §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 that 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).

As we will see, a number of elementary quadrature algorithms involve adding successive stages of refinement. It is convenient to encapsulate this feature in a `Quadrature` structure:

quadrature.h
```
struct Quadrature{
```
Abstract base class for elementary quadrature algorithms.
```
    Int n;
```
Current level of refinement.
```
    virtual Doub next() = 0;
```
Returns the value of the integral at the nth stage of refinement. The function `next()` must be defined in the derived class.
```
};
```

Then the `Trapzd` structure is derived from this as follows:

quadrature.h
```
template<class T>
struct Trapzd : Quadrature {
```
Routine implementing the extended trapezoidal rule.
```
    Doub a,b,s;
```
Limits of integration and current value of integral.
```
    T &func;
    Trapzd() {};
    Trapzd(T &funcc, const Doub aa, const Doub bb) :
        func(funcc), a(aa), b(bb) {n=0;}
```
The constructor takes as inputs `func`, the function or functor to be integrated between limits `a` and `b`, also input.
```
    Doub next() {
```
Returns the nth stage of refinement of the extended trapezoidal rule. On the first call (`n`=1), the routine returns the crudest estimate of $\int_a^b f(x)dx$. Subsequent calls set `n`=2,3,... and improve the accuracy by adding 2^{n-2} additional interior points.
```
        Doub x,tnm,sum,del;
        Int it,j;
        n++;
        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=0;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;
        }
    }
};
```

Note that `Trapzd` is templated on the whole struct and does not just contain a templated function. This is necessary because it retains a reference to the supplied function or functor as a member variable.

The `Trapzd` structure 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

```
    Ftor func;
```
Functor `func` here has no parameters.
```
    Trapzd<Ftor> s(func,a,b);
    for(j=1;j<=m+1;j++) val=s.next();
```

with the answer returned as `val`. Here `Ftor` is a functor containing the function to be integrated.

Much better, of course, is to refine the trapezoidal rule until some specified degree of accuracy has been achieved. A function for this is

quadrature.h

```
template<class T>
Doub qtrap(T &func, const Doub a, const Doub b, const Doub eps=1.0e-10) {
```

Returns the integral of the function or functor `func` from `a` to `b`. The constants `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.

```
    const Int JMAX=20;
    Doub s,olds=0.0;                    Initial value of olds is arbitrary.
    Trapzd<T> t(func,a,b);
    for (Int j=0;j<JMAX;j++) {
        s=t.next();
        if (j > 5)                      Avoid spurious early convergence.
            if (abs(s-olds) < eps*abs(olds) ||
                (s == 0.0 && olds == 0.0)) return s;
        olds=s;
    }
    throw("Too many steps in routine qtrap");
}
```

The optional argument `eps` sets the desired fractional accuracy. 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 that 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 of 10^{-10} or even smaller is usually no problem in double precision when the convergence is moderately rapid, but not otherwise. (Of course, very few problems really require such precision.)

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*,

$$\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{1}{2}f_0 + f_1 + f_2 + \cdots + f_{N-2} + \frac{1}{2}f_{N-1}\right] - \frac{B_2 h^2}{2!}(f'_{N-1} - f'_0) - \cdots - \frac{B_{2k}h^{2k}}{(2k)!}(f^{(2k-1)}_{N-1} - f^{(2k-1)}_0) - \cdots \tag{4.2.1}$$

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 $B_1 = -1/2$)

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

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

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

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:

quadrature.h

```
template<class T>
Doub qsimp(T &func, const Doub a, const Doub b, const Doub eps=1.0e-10) {
```

Returns the integral of the function or functor `func` from `a` to `b`. The constants `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.

```
    const Int JMAX=20;
    Doub s,st,ost=0.0,os=0.0;
    Trapzd<T> t(func,a,b);
    for (Int j=0;j<JMAX;j++) {
        st=t.next();
        s=(4.0*st-ost)/3.0;                Compare equation (4.2.4), above.
        if (j > 5)                         Avoid spurious early convergence.
            if (abs(s-os) < eps*abs(os) ||
                (s == 0.0 && os == 0.0)) return s;
        os=s;
        ost=st;
    }
    throw("Too many steps in routine qsimp");
}
```

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 fourth derivative (i.e., a continuous third 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. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), §3.1.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 2003 (New York: Dover), §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 §3.2) 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 a function call to Poly_interp::rawinterp, as given in §3.2.

romberg.h

```
template <class T>
Doub qromb(T &func, Doub a, Doub b, const Doub eps=1.0e-10) {
```

Returns the integral of the function or functor func from a to b. Integration is performed by Romberg's method of order 2K, where, e.g., K=2 is Simpson's rule.

```
    const Int JMAX=20, JMAXP=JMAX+1, 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.

```
    VecDoub s(JMAX),h(JMAXP);
    Poly_interp polint(h,s,K);
```

These store the successive trapezoidal approximations and their relative stepsizes.

```
    h[0]=1.0;
    Trapzd<T> t(func,a,b);
    for (Int j=1;j<=JMAX;j++) {
        s[j-1]=t.next();
        if (j >= K) {
            Doub ss=polint.rawinterp(j-K,0.0);
            if (abs(polint.dy) <= eps*abs(ss)) return ss;
        }
        h[j]=0.25*h[j-1];
```

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.

```
    }
    throw("Too many steps in routine qromb");
}
```

The routine qromb is quite powerful for sufficiently smooth (e.g., analytic) integrands, integrated over intervals that 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 second extrapolation, after just 6 calls to trapzd, while qsimp requires 11 calls (32 times as many evaluations of the integrand) and qtrap requires 19 calls (8192 times as many evaluations of the integrand).

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), §3.4 – §3.5.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 2003 (New York: Dover), §7.4.1 – §7.4.2.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), §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 19, notably §19.3. The fifth problem, singularity at an unknown location, can really only be handled by the use of a variable stepsize differential equation integration routine, as will be given in Chapter 17, or an adaptive quadrature routine such as in §4.7.

We need a workhorse like the extended trapezoidal rule (equation 4.1.11), but one that 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*,

$$\begin{aligned}\int_{x_0}^{x_{N-1}} f(x)dx = h[f_{1/2} + f_{3/2} + f_{5/2} + \cdots + f_{N-5/2} + f_{N-3/2}] \\ + \frac{B_2 h^2}{4}(f'_{N-1} - f'_0) + \cdots \\ + \frac{B_{2k} h^{2k}}{(2k)!}(1 - 2^{-2k+1})(f_{N-1}^{(2k-1)} - f_0^{(2k-1)}) + \cdots\end{aligned} \tag{4.4.1}$$

This equation can be derived by writing out (4.2.1) with stepsize h, then writing it out again with stepsize $h/2$, and 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 structure, which is directly comparable to `Trapzd`.

quadrature.h

```
template <class T>
struct Midpnt : Quadrature {
```

Routine implementing the extended midpoint rule.

```
    Doub a,b,s;
    T &funk;
    Midpnt(T &funcc, const Doub aa, const Doub bb) :
        funk(funcc), a(aa), b(bb) {n=0;}
```

Limits of integration and current value of integral.

The constructor takes as inputs `func`, the function or functor to be integrated between limits `a` and `b`, also input.

```
    Doub next(){
```

Returns the `n`th stage of refinement of the extended midpoint rule. On the first call (`n`=1), the routine returns the crudest estimate of $\int_a^b f(x)dx$. Subsequent calls set `n`=2,3,... and improve the accuracy by adding $(2/3) \times 3^{n-1}$ additional interior points.

```
        Int it,j;
        Doub x,tnm,sum,del,ddel;
        n++;
        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;
            x=a+0.5*del;
            sum=0.0;
            for (j=0;j<it;j++) {
                sum += func(x);
                x += ddel;
                sum += func(x);
                x += del;
            }
            s=(s+(b-a)*sum/tnm)/3.0;
            return s;
        }
    }
    virtual Doub func(const Doub x) {return funk(x);}
};
```

The added points alternate in spacing between `del` and `ddel`.

The new sum is combined with the old integral to give a refined integral.

Identity mapping.

You may have spotted a seemingly unnecessary extra level of indirection in `Midpnt`, namely its calling the user-supplied function `funk` through an identity function `func`. The reason for this is that we are going to use mappings other than the identity mapping between `funk` and `func` to solve the problems of improper integrals listed above. The new quadratures will simply be derived from `Midpnt` with `func` overridden.

The structure `Midpnt` could be used to exactly replace `Trapzd` in a driver routine like `qtrap` (§4.2); one could simply change `Trapzd<T> t(func,a,b)` to `Midpnt<T> t(func,a,b)`, and perhaps also decrease the parameter `JMAX` since

$3^{\mathtt{JMAX}-1}$ (from step tripling) is a much larger number than $2^{\mathtt{JMAX}-1}$ (step doubling). The open formula implementation analogous to Simpson's rule (`qsimp` in §4.2) could also substitute `Midpnt` for `Trapzd`, decreasing `JMAX` as above, but now also changing 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 thus modified `qtrap` or `qsimp` will fix the first problem on the list at the beginning of this section. More sophisticated, and allowing us to fix more problems, is to generalize Romberg integration in like manner:

romberg.h

```
template<class T>
Doub qromo(Midpnt<T> &q, const Doub eps=3.0e-9) {
```

Romberg integration on an open interval. Returns the integral of a function using any specified elementary quadrature algorithm q and Romberg's method. Normally q will be an open formula, not evaluating the function at the endpoints. It is assumed that q 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 q. The constants below have the same meanings as in `qromb`.

```
    const Int JMAX=14, JMAXP=JMAX+1, K=5;
    VecDoub h(JMAXP),s(JMAX);
    Poly_interp polint(h,s,K);
    h[0]=1.0;
    for (Int j=1;j<=JMAX;j++) {
        s[j-1]=q.next();
        if (j >= K) {
            Doub ss=polint.rawinterp(j-K,0.0);
            if (abs(polint.dy) <= eps*abs(ss)) return ss;
        }
        h[j]=h[j-1]/9.0;
    }
    throw("Too many steps in routine qromo");
}
```

This is where the assumption of step tripling and an even error series is used.

Notice that we now pass a `Midpnt` object instead of the user function and limits of integration. There is a good reason for this, as we will see below. It does, however, mean that you have to bind things together before calling `qromo`, something like this, where we integrate from `a` to `b`:

```
Midpnt<Ftor> q(ftor,a,b);
Doub integral=qromo(q);
```

or, for a bare function,

```
Midpnt<Doub(Doub)> q(fbare,a,b);
Doub integral=qromo(q);
```

Laid back C++ compilers will let you condense these to

```
Doub integral = qromo(Midpnt<Ftor>(Ftor(),a,b));
```

or

```
Doub integral = qromo(Midpnt<Doub(Doub)>(fbare,a,b));
```

but uptight compilers may object to the way that a temporary is passed by reference, in which case use the two-line forms above.

As we shall now see, the function `qromo`, with its peculiar interface, is an excellent driver routine for solving all the other problems of improper integrals in our first list (except the intractable fifth).

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 that decreases toward 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^{99}), or a to be negative and infinite. Since all the machinery is already in place in `Midpnt`, we write `Midinf` as a derived class and simply override the mapping function.

quadrature.h

```
template <class T>
struct Midinf : Midpnt<T>{
```

This routine is an exact replacement for `midpnt`, i.e., returns the `n`th stage of refinement of the integral of `funcc` 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.

```
    Doub func(const Doub x) {
        return Midpnt<T>::funk(1.0/x)/(x*x);        Effect the change of variable.
    }
    Midinf(T &funcc, const Doub aa, const Doub bb) :
        Midpnt<T>(funcc, aa, bb) {
        Midpnt<T>::a=1.0/bb;                        Set the limits of integration.
        Midpnt<T>::b=1.0/aa;
    }
};
```

An integral from 2 to ∞, for example, might be calculated by

```
Midinf<Ftor> q(ftor,2.,1.e99);
Doub integral=qromo(q);
```

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,

```
Midpnt<Ftor> q1(ftor,-5.,2.);
Midinf<Ftor> q2(ftor,2.,1.e99);
integral=qromo(q1)+qromo(q2);
```

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

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

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:

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

for a singularity at a, and

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

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

quadrature.h

```
template <class T>
struct Midsql : Midpnt<T>{
```

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`.

```
    Doub aorig;
    Doub func(const Doub x) {
        return 2.0*x*Midpnt<T>::funk(aorig+x*x);
    }
    Midsql(T &funcc, const Doub aa, const Doub bb) :
        Midpnt<T>(funcc, aa, bb), aorig(aa) {
        Midpnt<T>::a=0;
        Midpnt<T>::b=sqrt(bb-aa);
    }
};
```

Effect the change of variable.

Similarly,

quadrature.h

```
template <class T>
struct Midsqu : Midpnt<T>{
```

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`.

```
    Doub borig;
    Doub func(const Doub x) {
        return 2.0*x*Midpnt<T>::funk(borig-x*x);
    }
    Midsqu(T &funcc, const Doub aa, const Doub bb) :
        Midpnt<T>(funcc, aa, bb), borig(bb) {
        Midpnt<T>::a=0;
        Midpnt<T>::b=sqrt(bb-aa);
    }
};
```

Effect the change of variable.

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

quadrature.h

```
template <class T>
struct Midexp : Midpnt<T>{
```

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.

```
    Doub func(const Doub x) {
        return Midpnt<T>::funk(-log(x))/x;
    }
    Midexp(T &funcc, const Doub aa, const Doub bb) :
        Midpnt<T>(funcc, aa, bb) {
        Midpnt<T>::a=0.0;
        Midpnt<T>::b=exp(-aa);
    }
};
```

Effect the change of variable.

CITED REFERENCES AND FURTHER READING:

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

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall); reprinted 2003 (New York: Dover), §7.4.3, p. 294.

Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), §3.7.

4.5 Quadrature by Variable Transformation

Imagine a simple general quadrature algorithm that is very rapidly convergent and allows you to ignore endpoint singularities completely. Sound too good to be true? In this section we'll describe an algorithm that in fact handles large classes of integrals in exactly this way.

Consider evaluating the integral

$$I = \int_a^b f(x)dx \tag{4.5.1}$$

As we saw in the construction of equations (4.1.11) – (4.1.20), quadrature formulas of arbitrarily high order can be constructed with interior weights unity, just by tuning the weights near the endpoints. But if a function dies off rapidly enough near

the endpoints, then those weights don't matter at all. In such a case, an N-point quadrature with uniform weights converges converges exponentially with N. (For a more rigorous motivation of this idea, see §4.5.1. For the connection to Gaussian quadrature, see the discussion at the end of §20.7.4.)

What about a function that doesn't vanish at the endpoints? Consider a change of variables $x = x(t)$, such that $x \in [a, b] \to t \in [c, d]$:

$$I = \int_c^d f[x(t)] \frac{dx}{dt}\, dt \tag{4.5.2}$$

Choose the transformation such that the factor dx/dt goes rapidly to zero at the endpoints of the interval. Then the simple trapezoidal rule applied to (4.5.2) will give extremely accurate results. (In this section, we'll call quadrature with uniform weights trapezoidal quadrature, with the understanding that it's a matter of taste whether you weight the endpoints with weight $1/2$ or 1, since they don't count anyway.)

Even when $f(x)$ has integrable singularities at the endpoints of the interval, their effect can be overwhelmed by a suitable transformation $x = x(t)$. One need not tailor the transformation to the specific nature of the singularity: We will discuss several transformations that are effective at obliterating just about any kind of endpoint singularity.

The first transformation of this kind was introduced by Schwartz [1] and has become known as the TANH rule:

$$\begin{aligned} x &= \frac{1}{2}(b+a) + \frac{1}{2}(b-a)\tanh t, \qquad x \in [a, b] \to t \in [-\infty, \infty] \\ \frac{dx}{dt} &= \frac{1}{2}(b-a)\,\mathrm{sech}^2\, t = \frac{2}{b-a}(b-x)(x-a) \end{aligned} \tag{4.5.3}$$

The sharp decrease of $\mathrm{sech}^2\, t$ as $t \to \pm\infty$ explains the efficiency of the algorithm and its ability to deal with singularities. Another similar algorithm is the IMT rule [2]. However, $x(t)$ for the IMT rule is not given by a simple analytic expression, and its performance is not too different from the TANH rule.

There are two kinds of errors to consider when using something like the TANH rule. The *discretization error* is just the truncation error because you are using the trapezoidal rule to approximate I. The *trimming error* is the result of truncating the infinite sum in the trapezoidal rule at a finite value of N. (Recall that the limits are now $\pm\infty$.) You might think that the sharper the decrease of dx/dt as $t \to \pm\infty$, the more efficient the algorithm. But if the decrease is too sharp, then the density of quadrature points near the center of the original interval $[a, b]$ is low and the discretization error is large. The optimal strategy is to try to arrange that the discretization and trimming errors are approximately equal.

For the TANH rule, Schwartz [1] showed that the discretization error is of order

$$\epsilon_d \sim e^{-2\pi w/h} \tag{4.5.4}$$

where w is the distance from the real axis to the nearest singularity of the integrand. There is a pole when $\mathrm{sech}^2\, t \to \infty$, i.e., when $t = \pm i\pi/2$. If there are no poles closer to the real axis in $f(x)$, then $w = \pi/2$. The trimming error, on the other hand, is

$$\epsilon_t \sim \mathrm{sech}^2\, t_N \sim e^{-2Nh} \tag{4.5.5}$$

Setting $\epsilon_d \sim \epsilon_t$, we find

$$h \sim \frac{\pi}{(2N)^{1/2}}, \qquad \epsilon \sim e^{-\pi(2N)^{1/2}} \tag{4.5.6}$$

as the optimum h and the corresponding error. Note that ϵ decreases with N faster than any power of N. If f is singular at the endpoints, this can modify equation (4.5.5) for ϵ_t. This usually results in the constant π in (4.5.6) being reduced. Rather than developing an algorithm where we try to estimate the optimal h for each integrand a priori, we recommend simple step doubling and testing for convergence. We expect convergence to set in for h around the value given by equation (4.5.6).

The TANH rule essentially uses an exponential mapping to achieve the desired rapid fall-off at infinity. On the theory that more is better, one can try repeating the procedure. This leads to the DE (double exponential) rule:

$$\begin{aligned} x &= \frac{1}{2}(b+a) + \frac{1}{2}(b-a)\tanh(c\sinh t), \qquad x \in [a,b] \to t \in [-\infty,\infty] \\ \frac{dx}{dt} &= \frac{1}{2}(b-a)\,\mathrm{sech}^2(c\sinh t)c\cosh t \sim \exp(-c\exp|t|) \quad \text{as} \quad |t| \to \infty \end{aligned} \tag{4.5.7}$$

Here the constant c is usually taken to be 1 or $\pi/2$. (Values larger than $\pi/2$ are not useful since $w = \pi/2$ for $0 < c \le \pi/2$, but w decreases rapidly for larger c.) By an analysis similar to equations (4.5.4) – (4.5.6), one can show that the optimal h and corresponding error for the DE rule are of order

$$h \sim \frac{\log(2\pi N w/c)}{N}, \qquad \epsilon \sim e^{-kN/\log N} \tag{4.5.8}$$

where k is a constant. The improved performance of the DE rule over the TANH rule indicated by comparing equations (4.5.6) and (4.5.8) is borne out in practice.

4.5.1 Exponential Convergence of the Trapezoidal Rule

The error in evaluating the integral (4.5.1) by the trapezoidal rule is given by the Euler-Maclaurin summation formula,

$$I \approx \frac{h}{2}[f(a)+f(b)]+h\sum_{j=1}^{N-1} f(a+jh)-\sum_{k=1}^{\infty}\frac{B_{2k}h^{2k}}{(2k)!}[f^{(2k-1)}(b)-f^{(2k-1)}(a)] \tag{4.5.9}$$

Note that this is in general an asymptotic expansion, not a convergent series. If all the derivatives of the function f vanish at the endpoints, then all the "correction terms" in equation (4.5.9) are zero. The error in this case is very small — it goes to zero with h faster than any power of h. We say that the method converges *exponentially*. The straight trapezoidal rule is thus an excellent method for integrating functions such as $\exp(-x^2)$ on $(-\infty,\infty)$, whose derivatives all vanish at the endpoints.

The class of transformations that will produce exponential convergence for a function whose derivatives do not all vanish at the endpoints is those for which dx/dt and all its derivatives go to zero at the endpoints of the interval. For functions with singularities at the endpoints, we require that $f(x)\,dx/dt$ and all its derivatives vanish at the endpoints. This is a more precise statement of "dx/dt goes rapidly to zero" given above.

4.5.2 Implementation

Implementing the DE rule is a little tricky. It's not a good idea to simply use `Trapzd` on the function $f(x)\,dx/dt$. First, the factor $\operatorname{sech}^2(c\sinh t)$ in equation (4.5.7) can overflow if sech is computed as $1/\cosh$. We follow [3] and avoid this by using the variable q defined by

$$q = e^{-2\sinh t} \tag{4.5.10}$$

(we take $c = 1$ for simplicity) so that

$$\frac{dx}{dt} = 2(b-a)\frac{q}{(1+q)^2}\cosh t \tag{4.5.11}$$

For large positive t, q just underflows harmlessly to zero. Negative t is handled by using the symmetry of the trapezoidal rule about the midpoint of the interval. We write

$$\begin{aligned} I &\simeq h\sum_{j=-N}^{N} f(x_j)\left.\frac{dx}{dt}\right|_j \\ &= h\left\{ f[(a+b)/2]\left.\frac{dx}{dt}\right|_0 + \sum_{j=1}^{N}[f(a+\delta_j)+f(b-\delta_j)]\left.\frac{dx}{dt}\right|_j \right\} \end{aligned} \tag{4.5.12}$$

where

$$\delta = b - x = (b-a)\frac{q}{1+q} \tag{4.5.13}$$

A second possible problem is that cancellation errors in computing $a+\delta$ or $b-\delta$ can cause the computed value of $f(x)$ to blow up near the endpoint singularities. To handle this, you should code the function $f(x)$ as a function of two arguments, $f(x,\delta)$. Then compute the singular part using δ directly. For example, code the function $x^{-\alpha}(1-x)^{-\beta}$ as $\delta^{-\alpha}(1-x)^{-\beta}$ near $x=0$ and $x^{-\alpha}\delta^{-\beta}$ near $x=1$. (See §6.10 for another example of a $f(x,\delta)$.) Accordingly, the routine `DErule` below expects the function f to have two arguments. If your function has no singularities, or the singularities are "mild" (e.g., no worse than logarithmic), you can ignore δ when coding $f(x,\delta)$ and code it as if it were just $f(x)$.

The routine `DErule` implements equation (4.5.12). It contains an argument $h_{\max}$ that corresponds to the upper limit for t. The first approximation to I is given by the first term on the right-hand side of (4.5.12) with $h = h_{\max}$. Subsequent refinements correspond to halving h as usual. We typically take $h_{\max} = 3.7$ in double precision, corresponding to $q = 3\times 10^{-18}$. This is generally adequate for "mild" singularities, like logarithms. If you want high accuracy for stronger singularities, you may have to increase $h_{\max}$. For example, for $1/\sqrt{x}$ you need $h_{\max} = 4.3$ to get full double precision. This corresponds to $q = 10^{-32} = (10^{-16})^2$, as you might expect.

derule.h

```
template<class T>
struct DErule : Quadrature {
```

Structure for implementing the DE rule.

```
    Doub a,b,hmax,s;
    T &func;
```

```
    DErule(T &funcc, const Doub aa, const Doub bb, const Doub hmaxx=3.7)
        : func(funcc), a(aa), b(bb), hmax(hmaxx) {n=0;}
```

Constructor. funcc is the function or functor that provides the function to be integrated between limits aa and bb, also input. The function operator in funcc takes two arguments, x and δ, as described in the text. The range of integration in the transformed variable t is $(-\texttt{hmaxx}, \texttt{hmaxx})$. Typical values of hmaxx are 3.7 for logarithmic or milder singularities, and 4.3 for square-root singularities, as discussed in the text.

```
    Doub next() {
```

On the first call to the function next ($n = 1$), the routine returns the crudest estimate of $\int_a^b f(x)dx$. Subsequent calls to next ($n = 2, 3, \ldots$) will improve the accuracy by adding 2^{n-1} additional interior points.

```
        Doub del,fact,q,sum,t,twoh;
        Int it,j;
        n++;
        if (n == 1) {
            fact=0.25;
            return s=hmax*2.0*(b-a)*fact*func(0.5*(b+a),0.5*(b-a));
        } else {
            for (it=1,j=1;j<n-1;j++) it <<= 1;
            twoh=hmax/it;                    Twice the spacing of the points to be added.
            t=0.5*twoh;
            for (sum=0.0,j=0;j<it;j++) {
                q=exp(-2.0*sinh(t));
                del=(b-a)*q/(1.0+q);
                fact=q/SQR(1.0+q)*cosh(t);
                sum += fact*(func(a+del,del)+func(b-del,del));
                t += twoh;
            }
            return s=0.5*s+(b-a)*twoh*sum;  Replace s by its refined value and return.
        }
    }
};
```

If the double exponential rule (DE rule) is generally better than the single exponential rule (TANH rule), why don't we keep going and use a triple exponential rule, quadruple exponential rule, ...? As we mentioned earlier, the discretization error is dominated by the pole nearest to the real axis. It turns out that beyond the double exponential the poles come nearer and nearer to the real axis, so the methods tend to get worse, not better.

If the function to be integrated itself has a pole near the real axis (much nearer than the $\pi/2$ that comes from the DE or TANH rules), the convergence of the method slows down. In analytically tractable cases, one can find a "pole correction term" to add to the trapezoidal rule to restore rapid convergence [4].

4.5.3 Infinite Ranges

Simple variations of the TANH or DE rules can be used if either or both of the limits of integration is infinite:

Range	TANH Rule	DE Rule	Mixed Rule
$(0, \infty)$	$x = e^t$	$x = e^{2c \sinh t}$	$x = e^{t-e^{-t}}$
$(-\infty, \infty)$	$x = \sinh t$	$x = \sinh(c \sinh t)$	—

(4.5.14)

The last column gives a mixed rule for functions that fall off rapidly (e^{-x} or e^{-x^2}) at infinity. It is a DE rule at $x = 0$ but only a single exponential at infinity. The expo-

nential fall-off of the integrand makes it behave like a DE rule there too. The mixed rule for $(-\infty, \infty)$ is constructed by splitting the range into $(-\infty, 0)$ and $(0, \infty)$ and making the substitution $x \to -x$ in the first range. This gives two integrals on $(0, \infty)$.

To implement the DE rule for infinite ranges we don't need the precautions we used in coding the finite range DE rule. It's fine to simply use the routine `Trapzd` directly as a function of t, with the function `func` that it calls returning $f(x)\,dx/dt$. So if `funk` is your function returning $f(x)$, then you define the function `func` as a function of `t` by code of the following form (for the mixed rule)

```
x=exp(t-exp(-t));
dxdt=x*(1.0+exp(-t));
return funk(x)*dxdt;
```

and pass `func` to `Trapzd`. The only care required is in deciding the range of integration. You want the contribution to the integral from the endpoints of the integration to be negligible. For example, $(-4, 4)$ is typically adequate for $x = \exp(\pi \sinh t)$.

4.5.4 Examples

As examples of the power of these methods, consider the following integrals:

$$\int_0^1 \log x \log(1-x)\,dx = 2 - \frac{\pi^2}{6} \tag{4.5.15}$$

$$\int_0^\infty \frac{1}{x^{1/2}(1+x)}\,dx = \pi \tag{4.5.16}$$

$$\int_0^\infty x^{-3/2} \sin\frac{x}{2} e^{-x}\,dx = [\pi(\sqrt{5}-2)]^{1/2} \tag{4.5.17}$$

$$\int_0^\infty x^{-2/7} e^{-x^2}\,dx = \tfrac{1}{2}\Gamma(\tfrac{5}{14}) \tag{4.5.18}$$

The integral (4.5.15) is easily handled by `DErule`. The routine converges to machine precision (10^{-16}) with about 30 function evaluations, completely unfazed by the singularities at the endpoints. The integral (4.5.16) is an example of an integrand that is singular at the origin and falls off slowly at infinity. The routine `Midinf` fails miserably because of the slow fall-off. Yet the transformation $x = \exp(\pi \sinh t)$ again gives machine precision in about 30 function evaluations, integrating t over the range $(-4, 4)$. By comparison, the transformation $x = e^t$ for t in the range $(-90, 90)$ requires about 500 function evaluations for the same accuracy.

The integral (4.5.17) combines a singularity at the origin with exponential fall-off at infinity. Here the "mixed" transformation $x = \exp(t - e^{-t})$ is best, requiring about 60 function evaluations for t in the range $(-4.5, 4)$. Note that the exponential fall-off is crucial here; these transformations fail completely for slowly decaying oscillatory functions like $x^{-3/2} \sin x$. Fortunately the series acceleration algorithms of §5.3 work well in such cases.

The final integral (4.5.18) is similar to (4.5.17), and using the same transformation requires about the same number of function evaluations to achieve machine precision. The range of t can be smaller, say $(-4, 3)$, because of the more rapid fall-off of the integrand. Note that for all these integrals the number of function evaluations would be double the number we quote if we are using step doubling to

decide when the integrals have converged, since we need one extra set of trapezoidal evaluations to confirm convergence. In many cases, however, you don't need this extra set of function evaluations: Once the method starts converging, the number of significant digits approximately doubles with each iteration. Accordingly, you can set the convergence criterion to stop the procedure when two successive iterations agree to the *square root* of the desired precision. The last iteration will then have approximately the required precision. Even without this trick, the method is quite remarkable for the range of difficult integrals that it can tame efficiently.

An extended example of the use of the DE rule for finite and infinite ranges is given in §6.10. There we give a routine for computing the generalized Fermi-Dirac integrals

$$F_k(\eta,\theta) = \int_0^\infty \frac{x^k(1+\frac{1}{2}\theta x)^{1/2}}{e^{x-\eta}+1}\,dx \tag{4.5.19}$$

Another example is given in the routine `Stiel` in §4.6.

4.5.5 Relation to the Sampling Theorem

The *sinc expansion* of a function is

$$f(x) \simeq \sum_{k=-\infty}^{\infty} f(kh)\,\mathrm{sinc}\left[\frac{\pi}{h}(x-kh)\right] \tag{4.5.20}$$

where $\mathrm{sinc}(x) \equiv \sin x/x$. The expansion is exact for a limited class of analytic functions. However, it can be a good approximation for other functions too, and the sampling theorem characterizes these functions, as will be discussed in §13.11. There we will use the sinc expansion of e^{-x^2} to get an approximation for the complex error function. Functions well-approximated by the sinc expansion typically fall off rapidly as $x \to \pm\infty$, so truncating the expansion at $k = \pm N$ still gives a good approximation to $f(x)$.

If we integrate both sides of equation (4.5.20), we find

$$\int_{-\infty}^{\infty} f(x)\,dx \simeq h\sum_{k=-\infty}^{\infty} f(kh) \tag{4.5.21}$$

which is just the trapezoidal formula! Thus, rapid convergence of the trapezoidal formula for the integral of f corresponds to f being well-approximated by its sinc expansion. The various transformations described earlier can be used to map $x \to x(t)$ and produce good sinc approximations with uniform samples in t. These approximations can be used not only for the trapezoidal quadrature of f, but also for good approximations to derivatives, integral transforms, Cauchy principal value integrals, and solving differential and integral equations [5].

CITED REFERENCES AND FURTHER READING:

Schwartz, C. 1969, "Numerical Integration of Analytic Functions," *Journal of Computational Physics*, vol. 4, pp. 19–29.[1]

Iri, M., Moriguti, S., and Takasawa, Y. 1987, "On a Certain Quadrature Formula," *Journal of Computational and Applied Mathematics*, vol. 17, pp. 3–20. (English version of Japanese article originally published in 1970.)[2]

Evans, G.A., Forbes, R.C., and Hyslop, J. 1984, "The Tanh Transformation for Singular Integrals," *International Journal of Computer Mathematics*, vol. 15, pp. 339–358.[3]

Bialecki, B. 1989, *BIT*, "A Modified Sinc Quadrature Rule for Functions with Poles near the Arc of Integration," vol. 29, pp. 464–476.[4]

Stenger, F. 1981, "Numerical Methods Based on Whittaker Cardinal or Sinc Functions," *SIAM Review*, vol. 23, pp. 165–224.[5]

Takahasi, H., and Mori, H. 1973, "Quadrature Formulas Obtained by Variable Transformation," *Numerische Mathematik*, vol. 21, pp. 206–219.

Mori, M. 1985, "Quadrature Formulas Obtained by Variable Transformation and DE Rule," *Journal of Computational and Applied Mathematics*, vol. 12&13, pp. 119–130.

Sikorski, K., and Stenger, F. 1984, "Optimal Quadratures in H_p Spaces," *ACM Transactions on Mathematical Software*, vol. 10, pp. 140–151; *op. cit.*, pp. 152–160.

4.6 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

$$\int_a^b W(x) f(x) dx \approx \sum_{j=0}^{N-1} w_j f(x_j) \tag{4.6.1}$$

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

$$\int_{-1}^{1} \frac{\exp(-\cos^2 x)}{\sqrt{1-x^2}} dx \tag{4.6.2}$$

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

$$W(x) = \frac{1}{\sqrt{1-x^2}} \tag{4.6.3}$$

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 (4.6.1) 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 (4.6.1) becomes

$$\int_a^b g(x)dx \approx \sum_{j=0}^{N-1} v_j g(x_j) \tag{4.6.4}$$

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 (4.6.1), or like (4.6.4).

So far our introduction to Gaussian quadrature is pretty standard. However, there is an aspect of the method that is not as widely appreciated as it should be: For smooth integrands (after factoring out the appropriate weight function), Gaussian quadrature converges *exponentially* fast as N increases, because the order of the method, not just the density of points, increases with N. This behavior should be contrasted with the power-law behavior (e.g., $1/N^2$ or $1/N^4$) of the Newton-Cotes based methods in which the order remains fixed (e.g., 2 or 4) even as the density of points increases. For a more rigorous discussion, see §20.7.4.

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:

qgaus.h

```
template <class T>
Doub qgaus(T &func, const Doub a, const Doub b)
```

Returns the integral of the function or functor `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.

```
{
    Here are the abscissas and weights:
    static const Doub x[]={0.1488743389816312,0.4333953941292472,
        0.6794095682990244,0.8650633666889845,0.9739065285171717};
    static const Doub w[]={0.2955242247147529,0.2692667193099963,
        0.2190863625159821,0.1494513491505806,0.0666713443086881};
    Doub xm=0.5*(b+a);
    Doub xr=0.5*(b-a);
    Doub s=0;                          Will be twice the average value of the function, since the
    for (Int j=0;j<5;j++) {                ten weights (five numbers above each used twice)
        Doub dx=xr*x[j];                   sum to 2.
        s += w[j]*(func(xm+dx)+func(xm-dx));
    }
    return s *= xr;                    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

$$\langle f|g\rangle \equiv \int_a^b W(x)f(x)g(x)dx \tag{4.6.5}$$

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

$$\begin{aligned} p_{-1}(x) &\equiv 0 \\ p_0(x) &\equiv 1 \\ p_{j+1}(x) &= (x - a_j)p_j(x) - b_j p_{j-1}(x) \qquad j = 0, 1, 2, \ldots \end{aligned} \tag{4.6.6}$$

where

$$\begin{aligned} a_j &= \frac{\langle xp_j|p_j\rangle}{\langle p_j|p_j\rangle} \qquad j = 0, 1, \ldots \\ b_j &= \frac{\langle p_j|p_j\rangle}{\langle p_{j-1}|p_{j-1}\rangle} \qquad j = 1, 2, \ldots \end{aligned} \tag{4.6.7}$$

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

The polynomials defined by (4.6.6) are *monic*, that is, 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 (4.6.6) 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 (4.6.1) and (4.6.4) 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 it lets you find the abscissas for any particular case.

Once you know the abscissas $x_0, \ldots, x_{N-1}$, you need to find the weights w_j, $j = 0, \ldots, N-1$. One way to do this (not the most efficient) is to solve the set of linear equations

$$\begin{bmatrix} p_0(x_0) & \cdots & p_0(x_{N-1}) \\ p_1(x_0) & \cdots & p_1(x_{N-1}) \\ \vdots & & \vdots \\ p_{N-1}(x_0) & \cdots & p_{N-1}(x_{N-1}) \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{N-1} \end{bmatrix} = \begin{bmatrix} \int_a^b W(x)p_0(x)dx \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{4.6.8}$$

Equation (4.6.8) simply solves for those weights such that the quadrature (4.6.1) gives the correct answer for the integral of the first N orthogonal polynomials. Note that the zeros on the right-hand side of (4.6.8) 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

$$w_j = \frac{\langle p_{N-1} | p_{N-1} \rangle}{p_{N-1}(x_j)p'_N(x_j)} \tag{4.6.9}$$

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 (4.6.6), and (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 4.6.10 – 4.6.14 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.

4.6.1 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

(4.6.9). 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:

$$\begin{aligned} W(x) &= 1 \qquad -1 < x < 1 \\ (j+1)P_{j+1} &= (2j+1)xP_j - jP_{j-1} \end{aligned} \tag{4.6.10}$$

Gauss-Chebyshev:

$$\begin{aligned} W(x) &= (1-x^2)^{-1/2} \qquad -1 < x < 1 \\ T_{j+1} &= 2xT_j - T_{j-1} \end{aligned} \tag{4.6.11}$$

Gauss-Laguerre:

$$\begin{aligned} W(x) &= x^\alpha e^{-x} \qquad 0 < x < \infty \\ (j+1)L^\alpha_{j+1} &= (-x+2j+\alpha+1)L^\alpha_j - (j+\alpha)L^\alpha_{j-1} \end{aligned} \tag{4.6.12}$$

Gauss-Hermite:

$$\begin{aligned} W(x) &= e^{-x^2} \qquad -\infty < x < \infty \\ H_{j+1} &= 2xH_j - 2jH_{j-1} \end{aligned} \tag{4.6.13}$$

Gauss-Jacobi:

$$\begin{aligned} W(x) &= (1-x)^\alpha(1+x)^\beta \qquad -1 < x < 1 \\ c_j P^{(\alpha,\beta)}_{j+1} &= (d_j + e_j x)P^{(\alpha,\beta)}_j - f_j P^{(\alpha,\beta)}_{j-1} \end{aligned} \tag{4.6.14}$$

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

$$\begin{aligned} c_j &= 2(j+1)(j+\alpha+\beta+1)(2j+\alpha+\beta) \\ d_j &= (2j+\alpha+\beta+1)(\alpha^2-\beta^2) \\ e_j &= (2j+\alpha+\beta)(2j+\alpha+\beta+1)(2j+\alpha+\beta+2) \\ f_j &= 2(j+\alpha)(j+\beta)(2j+\alpha+\beta+2) \end{aligned} \tag{4.6.15}$$

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 (4.6.9) in the special form for the Gauss-Legendre case,

$$w_j = \frac{2}{(1-x_j^2)[P'_N(x_j)]^2} \tag{4.6.16}$$

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

$$\int_{x_1}^{x_2} f(x)dx = \sum_{j=0}^{N-1} w_j f(x_j) \tag{4.6.17}$$

gauss_wgts.h

```
void gauleg(const Doub x1, const Doub x2, VecDoub_O &x, VecDoub_O &w)
```

Given the lower and upper limits of integration x1 and x2, this routine returns arrays x[0..n-1] and w[0..n-1] of length n, containing the abscissas and weights of the Gauss-Legendre n-point quadrature formula.

```
{
    const Doub EPS=1.0e-14;                 EPS is the relative precision.
    Doub z1,z,xm,xl,pp,p3,p2,p1;
    Int n=x.size();
    Int 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 (Int i=0;i<m;i++) {                 Loop over the desired roots.
        z=cos(3.141592654*(i+0.75)/(n+0.5));
        Starting with this approximation to the ith root, we enter the main loop of refinement
        by Newton's method.
        do {
            p1=1.0;
            p2=0.0;
            for (Int j=0;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*p3)/(j+1);
            }
            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 (abs(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

$$\int_0^\infty x^\alpha e^{-x} f(x)dx = \sum_{j=0}^{N-1} w_j f(x_j) \tag{4.6.18}$$

gauss_wgts.h

```
void gaulag(VecDoub_O &x, VecDoub_O &w, const Doub alf)
```

Given alf, the parameter α of the Laguerre polynomials, this routine returns arrays x[0..n-1] and w[0..n-1] containing the abscissas and weights of the n-point Gauss-Laguerre quadrature formula. The smallest abscissa is returned in x[0], the largest in x[n-1].

```
{
    const Int MAXIT=10;
    const Doub EPS=1.0e-14;                 EPS is the relative precision.
    Int i,its,j;
    Doub ai,p1,p2,p3,pp,z,z1;
    Int n=x.size();
    for (i=0;i<n;i++) {                     Loop over the desired roots.
        if (i == 0) {                       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 == 1) {                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-1;
```

```
            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=0;its<MAXIT;its++) {          Refinement by Newton's method.
            p1=1.0;
            p2=0.0;
            for (j=0;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+alf)*p3)/(j+1);
            }
            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 (abs(z-z1) <= EPS) break;
        }
        if (its >= MAXIT) throw("too many iterations in gaulag");
        x[i]=z;                                Store the root and the weight.
        w[i] = -exp(gammln(alf+n)-gammln(Doub(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 (4.6.13), 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

$$\tilde{H}_{-1} = 0, \quad \tilde{H}_0 = \frac{1}{\pi^{1/4}}, \quad \tilde{H}_{j+1} = x\sqrt{\frac{2}{j+1}}\tilde{H}_j - \sqrt{\frac{j}{j+1}}\tilde{H}_{j-1} \tag{4.6.19}$$

The formula for the weights becomes

$$w_j = \frac{2}{[\tilde{H}'_N(x_j)]^2} \tag{4.6.20}$$

while the formula for the derivative with this normalization is

$$\tilde{H}'_j = \sqrt{2j}\,\tilde{H}_{j-1} \tag{4.6.21}$$

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

$$\int_{-\infty}^{\infty} e^{-x^2} f(x)dx = \sum_{j=0}^{N-1} w_j f(x_j) \tag{4.6.22}$$

gauss_wgts.h

```
void gauher(VecDoub_O &x, VecDoub_O &w)
```
This routine returns arrays `x[0..n-1]` and `w[0..n-1]` containing the abscissas and weights of the `n`-point Gauss-Hermite quadrature formula. The largest abscissa is returned in `x[0]`, the most negative in `x[n-1]`.
```
{
    const Doub EPS=1.0e-14,PIM4=0.7511255444649425;
    Relative precision and 1/π^{1/4}.
    const Int MAXIT=10;                        Maximum iterations.
    Int i,its,j,m;
```

```
    Doub p1,p2,p3,pp,z,z1;
    Int n=x.size();
    m=(n+1)/2;
    The roots are symmetric about the origin, so we have to find only half of them.
    for (i=0;i<m;i++) {                          Loop over the desired roots.
        if (i == 0) {                            Initial guess for the largest root.
            z=sqrt(Doub(2*n+1))-1.85575*pow(Doub(2*n+1),-0.16667);
        } else if (i == 1) {                     Initial guess for the second largest root.
            z -= 1.14*pow(Doub(n),0.426)/z;
        } else if (i == 2) {                     Initial guess for the third largest root.
            z=1.86*z-0.86*x[0];
        } else if (i == 3) {                     Initial guess for the fourth largest root.
            z=1.91*z-0.91*x[1];
        } else {                                 Initial guess for the other roots.
            z=2.0*z-x[i-2];
        }
        for (its=0;its<MAXIT;its++) {            Refinement by Newton's method.
            p1=PIM4;
            p2=0.0;
            for (j=0;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+1))*p2-sqrt(Doub(j)/(j+1))*p3;
            }
            p1 is now the desired Hermite polynomial. We next compute pp, its derivative, by
            the relation (4.6.21) using p2, the polynomial of one lower order.
            pp=sqrt(Doub(2*n))*p2;
            z1=z;
            z=z1-p1/pp;                          Newton's formula.
            if (abs(z-z1) <= EPS) break;
        }
        if (its >= MAXIT) throw("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

$$\int_{-1}^{1}(1-x)^{\alpha}(1+x)^{\beta}f(x)dx=\sum_{j=0}^{N-1}w_j f(x_j) \tag{4.6.23}$$

gauss_wgts.h

```
void gaujac(VecDoub_O &x, VecDoub_O &w, const Doub alf, const Doub bet)
```

Given `alf` and `bet`, the parameters α and β of the Jacobi polynomials, this routine returns arrays `x[0..n-1]` and `w[0..n-1]` containing the abscissas and weights of the `n`-point Gauss-Jacobi quadrature formula. The largest abscissa is returned in `x[0]`, the smallest in `x[n-1]`.

```
{
    const Int MAXIT=10;
    const Doub EPS=1.0e-14;                  EPS is the relative precision.
    Int i,its,j;
    Doub alfbet,an,bn,r1,r2,r3;
    Doub a,b,c,p1,p2,p3,pp,temp,z,z1;
    Int n=x.size();
    for (i=0;i<n;i++) {                      Loop over the desired roots.
        if (i == 0) {                        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 == 1) {                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*abs(alf))/n;
            z -= (1.0-z)*r1*r2*r3;
        } else if (i == 2) {                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[0]-z)*r1*r2*r3;
        } else if (i == n-2) {              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-4])*r1*r2*r3;
        } else if (i == n-1) {              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-3])*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 (abs(z-z1) <= EPS) break;
        }
        if (its > MAXIT) throw("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:

$$x_j = \cos\left(\frac{\pi(j+\frac{1}{2})}{N}\right)$$
$$w_j = \frac{\pi}{N} \tag{4.6.24}$$

4.6.2 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 (4.6.9) above, since the derivative p'_N can be efficiently computed by the derivative of (4.6.6) in the general case, or by special relations for the classical polynomials. Note that (4.6.9) 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 (4.6.6) and the term p_{j+1} to the right-hand side, the recurrence relation can be written in matrix form as

$$x\begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{N-2} \\ p_{N-1} \end{bmatrix} = \begin{bmatrix} a_0 & 1 & & & & \\ b_1 & a_1 & 1 & & & \\ & \vdots & \vdots & & & \\ & & & b_{N-2} & a_{N-2} & 1 \\ & & & & b_{N-1} & a_{N-1} \end{bmatrix} \cdot \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{N-2} \\ p_{N-1} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ p_N \end{bmatrix} \tag{4.6.25}$$

or

$$x\mathbf{p} = \mathbf{T}\cdot\mathbf{p} + p_N\mathbf{e}_{N-1} \tag{4.6.26}$$

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

$$\mathbf{J} = \mathbf{DTD}^{-1} = \begin{bmatrix} a_0 & \sqrt{b_1} & & & \\ \sqrt{b_1} & a_1 & \sqrt{b_2} & & \\ & \vdots & \vdots & & \\ & & \sqrt{b_{N-2}} & a_{N-2} & \sqrt{b_{N-1}} \\ & & & \sqrt{b_{N-1}} & a_{N-1} \end{bmatrix} \tag{4.6.27}$$

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 (4.6.26) 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

$$w_j = \mu_0 v_{j,0}^2 \tag{4.6.28}$$

where

$$\mu_0 = \int_a^b W(x)\,dx \tag{4.6.29}$$

and where $v_{j,0}$ is the zeroth 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.

```
void gaucof(VecDoub_IO &a, VecDoub_IO &b, const Doub amu0, VecDoub_O &x,
    VecDoub_O &w)
```
gauss_wgts2.h

Computes the abscissas and weights for a Gaussian quadrature formula from the Jacobi matrix. On input, `a[0..n-1]` and `b[0..n-1]` are the coefficients of the recurrence relation for the set of monic orthogonal polynomials. The quantity $\mu_0 \equiv \int_a^b W(x)\,dx$ is input as `amu0`. The abscissas `x[0..n-1]` are returned in descending order, with the corresponding weights in `w[0..n-1]`. The arrays `a` and `b` are modified. Execution can be speeded up by modifying `tqli` and `eigsrt` to compute only the zeroth component of each eigenvector.

```
{
    Int n=a.size();
    for (Int i=0;i<n;i++)
        if (i != 0) b[i]=sqrt(b[i]);          Set up superdiagonal of Jacobi matrix.
    Symmeig sym(a,b);
    for (Int i=0;i<n;i++) {
        x[i]=sym.d[i];
        w[i]=amu0*sym.z[0][i]*sym.z[0][i];    Equation (4.6.28).
    }
}
```

4.6.3 Orthogonal Polynomials with Nonclassical Weights

What do you 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 (4.6.6) to use in `gaucof`? Obviously, you need a method of finding the a_j's and b_j's.

The best general method is the *Stieltjes procedure*: First compute a_0 from (4.6.7), and then $p_1(x)$ from (4.6.6). Knowing p_0 and p_1, compute a_1 and b_1 from (4.6.7), and so on. But how are we to compute the inner products in (4.6.7)?

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,

$$\mu_j = \int_a^b x^j W(x)dx \qquad j = 0, 1, \ldots, 2N-1 \tag{4.6.30}$$

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 (4.6.30).

Gautschi [5] showed that the Stieltjes procedure is feasible if the inner products in (4.6.7) are computed directly by numerical quadrature. This is only practicable if you can find a quadrature scheme that can compute the integrals to high accuracy despite the singularities in the weight function $W(x)$. Gautschi advocates the Fejér quadrature scheme [5] as a general-purpose scheme for handling singularities when no better method is available. We have personally had much better experience with the transformation methods of §4.5, particularly the DE rule and its variants.

We use a structure `Stiel` that implements the Stieltjes procedure. Its member function `get_weights` generates the coefficients a_j and b_j of the recurrence relation, and then calls `gaucof` to find the abscissas and weights. You can easily modify it to return the a_j's and b_j's if you want them as well. Internally, the routine calls the function `quad` to do the integrals in (4.6.7). For a finite range of integration, the routine uses the straight DE rule. This is effected by invoking the constructor with five parameters: the number of quadrature abscissas (and weights) desired, the lower and upper limits of integration, the parameter $h_{\max}$ to be passed to the DE rule (see §4.5), and the weight function $W(x)$. For an infinite range of integration, the routine invokes the trapezoidal rule with one of the coordinate transformations discussed in §4.5. For this case you invoke the constructor that has no $h_{\max}$, but takes the mapping function $x = x(t)$ and its derivative dx/dt in addition to $W(x)$. Now the range of integration you input is the finite range of the trapezoidal rule.

This will all be clearer with some examples. Consider first the weight function

$$W(x) = -\log x \tag{4.6.31}$$

on the finite interval $(0, 1)$. Normally, for the finite range case (DE rule), the weight function must be coded as a function of two variables, $W(x, \delta)$, where δ is the distance from the endpoint singularity. Since the logarithmic singularity at the endpoint $x = 0$ is "mild," there is no need to use the argument δ in coding the function:

```
Doub wt(const Doub x, const Doub del)
{
    return -log(x);
}
```

A value of $h_{\max} = 3.7$ will give full double precision, as discussed in §4.5, so the calling code looks like this:

```
n= ...
VecDoub x(n),w(n);
Stiel s(n,0.0,1.0,3.7,wt);
s.get_weights(x,w);
```

For the infinite range case, in addition to the weight function $W(x)$, you have to supply two functions for the coordinate transformation you want to use (see equation 4.5.14). We'll denote the mapping $x = x(t)$ by `fx` and dx/dt by `fdxdt`, but you can use any names you like. All these functions are coded as functions of one variable.

Here is an example of the user-supplied functions for the weight function

$$W(x) = \frac{x^{1/2}}{e^x + 1} \tag{4.6.32}$$

on the interval $(0, \infty)$. Gaussian quadrature based on $W(x)$ has been proposed for evaluating generalized Fermi-Dirac integrals [6] (cf. §4.5). We use the "mixed" DE rule of equation (4.5.14), $x = e^{t-e^{-t}}$. As is typical with the Stieltjes procedure, you get abscissas and weights within about one or two significant digits of machine accuracy for N of a few dozen.

```
Doub wt(const Doub x)
{
    Doub s=exp(-x);
    return sqrt(x)*s/(1.0+s);
}

Doub fx(const Doub t)
{
    return exp(t-exp(-t));
}

Doub fdxdt(const Doub t)
{
    Doub s=exp(-t);
    return exp(t-s)*(1.0+s);
}
    ...
Stiel ss(n,-5.5,6.5,wt,fx,fdxdt);
ss.get_weights(x,w);
```

The listing of the `Stiel` object, and discussion of some of the C++ intricacies of its coding, are in a Webnote [9].

Two other algorithms exist [7,8] for finding abscissas and weights for Gaussian quadratures. The first starts similarly to the Stieltjes procedure by representing the inner product integrals in equation (4.6.7) as discrete quadratures using some quadrature rule. This defines a matrix whose elements are formed from the abscissas and weights in your chosen quadrature rule, together with the given weight function. Then an algorithm due to Lanczos is used to transform this to a matrix that is essentially the Jacobi matrix (4.6.27).

The second algorithm is based on the idea of *modified moments*. 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. Then the inner products in equation (4.6.7) will be expressible

in terms of the modified moments

$$\nu_j = \int_a^b \pi_j(x)W(x)dx \qquad j = 0, 1, \ldots, 2N-1 \tag{4.6.33}$$

The *modified Chebyshev algorithm* (due to Sack and Donovan [10] and later improved by Wheeler [11]) is an efficient algorithm that generates the desired a_j's and b_j's from the modified moments. 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 $W(x)$. The algorithm requires that the modified moments (4.6.33) be accurately computed. Sometimes there is a closed form, for example, for the important case of the $\log x$ weight function [12,8]. Otherwise you have to use a suitable discretization procedure to compute the modified moments [7,8], just as we did for the inner products in the Stieltjes procedure. There is some art in choosing the auxiliary polynomials π_j, and in practice it is not always possible to find a set that removes the ill-conditioning.

Gautschi [8] has given an extensive suite of routines that handle all three of the algorithms we have described, together with many other aspects of orthogonal polynomials and Gaussian quadrature. However, for most straightforward applications, you should find `Stiel` together with a suitable DE rule quadrature more than adequate.

4.6.4 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 [13,8] has given an algorithm similar to `gaucof` for these cases.

An N-point Gauss-Radau rule has the form of equation (4.6.1), where x_1 is chosen to be either a or b (x_1 must be finite). You can construct the rule from the coefficients for the corresponding ordinary N-point Gaussian quadrature. Simply set up the Jacobi matrix equation (4.6.27), but modify the entry a_{N-1}:

$$a'_{N-1} = x_1 - b_{N-1}\frac{p_{N-2}(x_1)}{p_{N-1}(x_1)} \tag{4.6.34}$$

Here is the routine:

gauss_wgts2.h

```
void radau(VecDoub_IO &a, VecDoub_IO &b, const Doub amu0, const Doub x1,
    VecDoub_O &x, VecDoub_O &w)
```

Computes the abscissas and weights for a Gauss-Radau quadrature formula. On input, `a[0..n-1]` and `b[0..n-1]` are the coefficients of the recurrence relation for the set of monic orthogonal polynomials corresponding to the weight function. (`b[0]` is not referenced.) The quantity $\mu_0 \equiv \int_a^b W(x)\,dx$ is input as `amu0`. `x1` is input as either endpoint of the interval. The abscissas `x[0..n-1]` are returned in descending order, with the corresponding weights in `w[0..n-1]`. The arrays `a` and `b` are modified.

```
{
    Int n=a.size();
    if (n == 1) {
        x[0]=x1;
        w[0]=amu0;
    } else {
        Doub p=x1-a[0];
```

Compute p_{N-1} and p_{N-2} by recurrence.

```
        Doub pm1=1.0;
        Doub p1=p;
        for (Int i=1;i<n-1;i++) {
            p=(x1-a[i])*p1-b[i]*pm1;
            pm1=p1;
            p1=p;
        }
        a[n-1]=x1-b[n-1]*pm1/p;          Equation (4.6.34).
        gaucof(a,b,amu0,x,w);
    }
}
```

An N-point Gauss-Lobatto rule has the form of equation (4.6.1) where $x_1 = a$, $x_N = b$ (both finite). This time you modify the entries a_{N-1} and b_{N-1} in equation (4.6.27) by solving two linear equations:

$$\begin{bmatrix} p_{N-1}(x_1) & p_{N-2}(x_1) \\ p_{N-1}(x_N) & p_{N-2}(x_N) \end{bmatrix} \begin{bmatrix} a'_{N-1} \\ b'_{N-1} \end{bmatrix} = \begin{bmatrix} x_1 p_{N-1}(x_1) \\ x_N p_{N-1}(x_N) \end{bmatrix} \tag{4.6.35}$$

gauss_wgts2.h

```
void lobatto(VecDoub_IO &a, VecDoub_IO &b, const Doub amu0, const Doub x1,
    const Doub xn, VecDoub_O &x, VecDoub_O &w)
```

Computes the abscissas and weights for a Gauss-Lobatto quadrature formula. On input, the vectors `a[0..n-1]` and `b[0..n-1]` are the coefficients of the recurrence relation for the set of monic orthogonal polynomials corresponding to the weight function. (`b[0]` is not referenced.) The quantity $\mu_0 \equiv \int_a^b W(x)\,dx$ is input as `amu0`. `x1` amd `xn` are input as the endpoints of the interval. The abscissas `x[0..n-1]` are returned in descending order, with the corresponding weights in `w[0..n-1]`. The arrays `a` and `b` are modified.

```
{
    Doub det,pl,pr,p1l,p1r,pm1l,pm1r;
    Int n=a.size();
    if (n <= 1)
        throw("n must be bigger than 1 in lobatto");
    pl=x1-a[0];                          Compute $p_{N-1}$ and $p_{N-2}$ at $x_1$ and $x_N$ by recur-
    pr=xn-a[0];                              rence.
    pm1l=1.0;
    pm1r=1.0;
    p1l=pl;
    p1r=pr;
    for (Int i=1;i<n-1;i++) {
        pl=(x1-a[i])*p1l-b[i]*pm1l;
        pr=(xn-a[i])*p1r-b[i]*pm1r;
        pm1l=p1l;
        pm1r=p1r;
        p1l=pl;
        p1r=pr;
    }
    det=pl*pm1r-pr*pm1l;                 Solve equation (4.6.35).
    a[n-1]=(x1*pl*pm1r-xn*pr*pm1l)/det;
    b[n-1]=(xn-x1)*pl*pr/det;
    gaucof(a,b,amu0,x,w);
}
```

The second important extension of Gaussian quadrature 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 [14] 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 [15] showed how to compute continued extensions of this kind. Sequences such as $N = 10, 21, 43, 87, \ldots$ are popular in automatic quadrature routines [16] 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* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at `http://www.nr.com/aands`, §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, "Calculation of Gauss Quadrature Rules," *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]

Gautschi, W. 1968, "Construction of Gauss-Christoffel Quadrature Formulas," *Mathematics of Computation*, vol. 22, pp. 251–270.[5]

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

Gautschi, W. 1982, "On Generating Orthogonal Polynomials," *SIAM Journal on Scientific and Statistical Computing*, vol. 3, pp. 289–317.[7]

Gautschi, W. 1994, "ORTHPOL: A Package of Routines for Generating Orthogonal Polynomials and Gauss-type Quadrature Rules," *ACM Transactions on Mathematical Software*, vol. 20, pp. 21–62 (Algorithm 726 available from netlib).[8]

Numerical Recipes Software 2007, "Implementation of Stiel," *Numerical Recipes Webnote No. 3*, at `http://www.nr.com/webnotes?3` [9]

Sack, R.A., and Donovan, A.F. 1971/72, "An Algorithm for Gaussian Quadrature Given Modified Moments," *Numerische Mathematik*, vol. 18, pp. 465–478.[10]

Wheeler, J.C. 1974, "Modified Moments and Gaussian Quadratures," *Rocky Mountain Journal of Mathematics*, vol. 4, pp. 287–296.[11]

Gautschi, W. 1978, in *Recent Advances in Numerical Analysis*, C. de Boor and G.H. Golub, eds. (New York: Academic Press), pp. 45–72.[12]

Golub, G.H. 1973, "Some Modified Matrix Eigenvalue Problems," *SIAM Review*, vol. 15, pp. 318–334.[13]

Kronrod, A.S. 1964, *Doklady Akademii Nauk SSSR*, vol. 154, pp. 283–286 (in Russian); translated as *Soviet Physics "Doklady"*.[14]

Patterson, T.N.L. 1968, "The Optimum Addition of Points to Quadrature Formulae," *Mathematics of Computation*, vol. 22, pp. 847–856 and C1–C11; 1969, *op. cit.*, vol. 23, p. 892.[15]

Piessens, R., de Doncker-Kapenga, E., Überhuber, C., and Kahaner, D. 1983 *QUADPACK, A Subroutine Package for Automatic Integration* (New York: Springer). Software at `http://www.netlib.org/quadpack`.[16]

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

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

Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), §3.6.

4.7 Adaptive Quadrature

The idea behind adaptive quadrature is very simple. Suppose you have two different numerical estimates I_1 and I_2 of the integral

$$I = \int_a^b f(x)\,dx \tag{4.7.1}$$

Suppose I_1 is more accurate. Use the relative difference between I_1 and I_2 as an error estimate. If it is less than ϵ, accept I_1 as the answer. Otherwise divide the interval $[a, b]$ into two subintervals,

$$I = \int_a^m f(x)\,dx + \int_m^b f(x)\,dx \qquad m = (a+b)/2 \tag{4.7.2}$$

and compute the two integrals independently. For each one, compute an I_1 and I_2, estimate the error, and continue subdividing if necessary. Dividing any given subinterval stops when its contribution to ϵ is sufficiently small. (Obviously recursion will be a good way to implement this algorithm.)

The most important criterion for an adaptive quadrature routine is reliability: If you request an accuracy of 10^{-6}, you would like to be sure that the answer is at least that good. From a theoretical point of view, however, it is impossible to design an adaptive quadrature routine that will work for all possible functions. The reason is simple: A quadrature is based on the value of the integrand $f(x)$ at a *finite* set of points. You can alter the function at all the other points in an arbitrary way without affecting the estimate your algorithm returns, while the true value of the integral changes unpredictably. Despite this point of principle, however, in practice good routines are reliable for a high fraction of functions they encounter. Our favorite routine is one proposed by Gander and Gautschi [1], which we now describe. It is relatively simple, yet scores well on reliability and efficiency.

A key component of a good adaptive algorithm is the termination criterion. The usual criterion

$$|I_1 - I_2| < \epsilon |I_1| \tag{4.7.3}$$

is problematic. In the neighborhood of a singularity, I_1 and I_2 might never agree to the requested tolerance, even if it's not particularly small. Instead, you need to somehow come up with an estimate of the *whole* integral I of equation (4.7.1). Then you can terminate when the error in I_1 is negligible compared to the whole integral:

$$|I_1 - I_2| < \epsilon |I_s| \tag{4.7.4}$$

where I_s is the estimate of I. Gander and Gautschi implement this test by writing

```
if (is + (i1-i2) == is)
```

which is equivalent to setting ϵ to the machine precision. However, modern optimizing compilers have become too good at recognizing that this is algebraically equivalent to

```
if (i1-i2 == 0.0)
```

which might never be satisfied in floating point arithmetic. Accordingly, we implement the test with an explicit ϵ.

The other problem you need to take care of is when an interval gets subdivided so small that it contains no interior machine-representable point. You then need to terminate the recursion and alert the user that the full accuracy might not have been attained. In the case where the points in an interval are supposed to be $\{a, m = (a+b)/2, b\}$, you can test for $m \le a$ or $b \le m$.

The lowest order integration method in the Gander-Gautschi method is the four-point Gauss-Lobatto quadrature (cf. §4.6)

$$\int_{-1}^{1} f(x)\,dx = \tfrac{1}{6}\Big[f(-1)+f(1)\Big] + \tfrac{5}{6}\Big[f\Big(-\tfrac{1}{\sqrt{5}}\Big)+f\Big(\tfrac{1}{\sqrt{5}}\Big)\Big] \tag{4.7.5}$$

This formula, which is exact for polynomials of degree 5, is used to compute I_2. To reuse these function evaluations in computing I_1, they find the seven-point Kronrod extension,

$$\begin{aligned}\int_{-1}^{1} f(x)\,dx = {} & \tfrac{11}{210}\Big[f(-1)+f(1)\Big] + \tfrac{72}{245}\Big[f\Big(-\sqrt{\tfrac{2}{3}}\Big)+f\Big(\sqrt{\tfrac{2}{3}}\Big)\Big] \\ & + \tfrac{125}{294}\Big[f\Big(-\tfrac{1}{\sqrt{5}}\Big)+f\Big(\tfrac{1}{\sqrt{5}}\Big)\Big] + \tfrac{16}{35}f(0)\end{aligned} \tag{4.7.6}$$

whose degree of exactness is nine. The formulas (4.7.5) and (4.7.6) get scaled from $[-1,1]$ to an arbitrary subinterval $[a,b]$.

For I_s, Gander and Gautschi find a 13-point Kronrod extension of equation (4.7.6), which lets them reuse the previous function evaluations. The formula is coded into the routine below. You can think of this initial 13-point evaluation as a kind of Monte Carlo sampling to get an idea of the order of magnitude of the integral. But if the integrand is smooth, this initial evaluation will itself be quite accurate already. The routine below takes advantage of this.

Note that to reuse the four function evaluations in (4.7.5) in the seven-point formula (4.7.6), you can't simply bisect intervals. But dividing into six subintervals works (there are six intervals between seven points).

To use the routine, you need to initialize an `Adapt` object with your required tolerance,

```
Adapt s(1.0e-6);
```

and then call the `integrate` function:

```
ans=s.integrate(func,a,b);
```

You should check that the desired tolerance could be met:

```
if (s.out_of_tolerance)
    cout << "Required tolerance may not be met" << endl;
```

The smallest allowed tolerance is 10 times the machine precision. If you enter a smaller tolerance, it gets reset internally. (The routine will work using the machine precision itself, but then it usually just takes lots of function evaluations for little additional benefit.)

The implementation of the `Adapt` object is given in a Webnote [2].

Adaptive quadrature is no panacea. The above routine has no special machinery to deal with singularities other than to refine the neighboring intervals. By using

suitable schemes for I_1 and I_2, one can customize an adaptive routine to deal with a particular kind of singularity (cf. [3]).

CITED REFERENCES AND FURTHER READING:

Gander, W., and Gautschi, W. 2000, "Adaptive Quadrature — Revisited," *BIT* vol. 40, pp. 84–101.[1]

Numerical Recipes Software 2007, "Implementation of Adapt," *Numerical Recipes Webnote No. 4*, at `http://www.nr.com/webnotes?4` [2]

Piessens, R., de Doncker-Kapenga, E., Überhuber, C., and Kahaner, D. 1983 *QUADPACK, A Subroutine Package for Automatic Integration* (New York: Springer). Software at `http://www.netlib.org/quadpack`.[3]

Davis, P.J., and Rabinowitz, P. 1984, *Methods of Numerical Integration*, 2nd ed., (Orlando, FL: Academic Press), Chapter 6.

4.8 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 that 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

$$\int_0^x dt_n \int_0^{t_n} dt_{n-1} \cdots \int_0^{t_3} dt_2 \int_0^{t_2} f(t_1)\, dt_1 = \frac{1}{(n-1)!} \int_0^x (x-t)^{n-1} f(t)\, dt \tag{4.8.1}$$

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 §7.9.)

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

$$\begin{aligned} I &\equiv \iiint dx\, dy\, dz\, f(x, y, z) \\ &= \int_{x_1}^{x_2} dx \int_{y_1(x)}^{y_2(x)} dy \int_{z_1(x,y)}^{z_2(x,y)} dz\, f(x, y, z) \end{aligned} \tag{4.8.2}$$

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

$$\int_{-1}^{1} dx \int_{-\sqrt{1-x^2}}^{\sqrt{1-x^2}} dy\, f(x, y) \tag{4.8.3}$$

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

$$G(x, y) \equiv \int_{z_1(x,y)}^{z_2(x,y)} f(x, y, z)\, dz \tag{4.8.4}$$

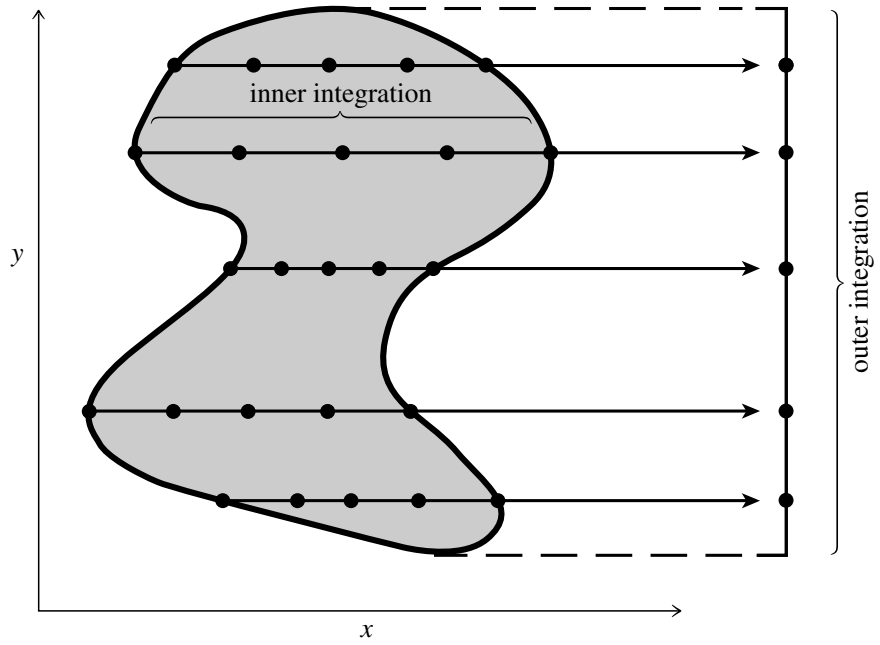

Figure 4.8.1. Function evaluations for a two-dimensional integral over an irregular region, shown schematically. The outer integration routine, in y, requests values of the inner, x, integral at locations along the y-axis of its own choosing. The inner integration routine then evaluates the function at x locations suitable to *it*. This is more accurate in general than, e.g., evaluating the function on a Cartesian mesh of points.

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

$$H(x) \equiv \int_{y_1(x)}^{y_2(x)} G(x, y)\, dy \tag{4.8.5}$$

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

$$I = \int_{x_1}^{x_2} H(x)\, dx \tag{4.8.6}$$

In an implementation of equations (4.8.4) – (4.8.6), 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 4.8.1). Current values of x and y, and the pointers to the user-supplied functions for the integrand and the boundaries, are passed "over the head" of the intermediate calls through member variables in the three functors defining the integrands for G, H and I.

quad3d.h

```
struct NRf3 {
    Doub xsav,ysav;
    Doub (*func3d)(const Doub, const Doub, const Doub);
    Doub operator()(const Doub z)
    {
        return func3d(xsav,ysav,z);
    }
};
```

The integrand $f(x, y, z)$ evaluated at fixed x and y.

```
struct NRf2 {
    NRf3 f3;
    Doub (*z1)(Doub, Doub);
    Doub (*z2)(Doub, Doub);
    NRf2(Doub zz1(Doub, Doub), Doub zz2(Doub, Doub)) : z1(zz1), z2(zz2) {}
    Doub operator()(const Doub y)   This is G of eq. (4.8.4).
    {
        f3.ysav=y;
        return qgaus(f3,z1(f3.xsav,y),z2(f3.xsav,y));
    }
};
struct NRf1 {
    Doub (*y1)(Doub);
    Doub (*y2)(Doub);
    NRf2 f2;
    NRf1(Doub yy1(Doub), Doub yy2(Doub), Doub z1(Doub, Doub),
        Doub z2(Doub, Doub)) : y1(yy1),y2(yy2), f2(z1,z2) {}
    Doub operator()(const Doub x)   This is H of eq. (4.8.5).
    {
        f2.f3.xsav=x;
        return qgaus(f2,y1(x),y2(x));
    }
};

template <class T>
Doub quad3d(T &func, const Doub x1, const Doub x2, Doub y1(Doub), Doub y2(Doub),
    Doub z1(Doub, Doub), Doub z2(Doub, Doub))
```

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 `y1`, `y2`, `z1`, and `z2`, as defined in (4.8.2). Integration is performed by calling `qgaus` recursively.

```
{
    NRf1 f1(y1,y2,z1,z2);
    f1.f2.f3.func3d=func;
    return qgaus(f1,x1,x2);
}
```

Note that while the function to be integrated can be supplied either as a simple function

```
Doub func(const Doub x, const Doub y, const Doub z);
```

or as the equivalent functor, the functions defining the boundary can only be functions:

```
Doub y1(const Doub x);
Doub y2(const Doub x);
Doub z1(const Doub x, const Doub y);
Doub z2(const Doub x, const Doub y);
```

This is for simplicity; you can easily modify the code to take functors if you need to.

The Gaussian quadrature routine used in `quad3d` is simple, but its accuracy is not controllable. An alternative is to use a one-dimensional integration routine like `qtrap`, `qsimp` or `qromb`, which have a user-definable tolerance `eps`. Simply replace all occurrences of `qgaus` in `quad3d` by `qromb`, say.

Note that multidimensional integration is likely to be very slow if you try for too much accuracy. You should almost certainly increase the default `eps` in `qromb` from 10^{-10} to 10^{-6} or bigger. You should also decrease `JMAX` to avoid a lot of waiting around for an answer. Some people advocate using a smaller `eps` for the inner quadrature (over z in our routine) than for the outer quadratures (over x or 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); reprinted 2003 (New York: Dover), §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* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at `http://www.nr.com/aands`, equations 25.4.58ff.