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

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

Numerical Recipes: The Art of Scientific Computing, Third Edition, by W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. Version 3.04 (2011). Text is Copyright © 1988-2007 by Cambridge University Press. Computer source code is Copyright © 1987-2007 by Numerical Recipes Software. Hardcover book ISBN 978-0-521-88068-8 is published by Cambridge University Press, http://www.cambridge.org. This electronic edition is published by Numerical Recipes Software, http://numerical.recipes. Permission is granted for authorized subscribers to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any public server computer, is strictly prohibited.

(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.

gauss_wgts2.h

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

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

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 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://numerical.recipes/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://numerical.recipes/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

Numerical Recipes: The Art of Scientific Computing, Third Edition, by W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. Version 3.04 (2011). Text is Copyright © 1988-2007 by Cambridge University Press. Computer source code is Copyright © 1987-2007 by Numerical Recipes Software. Hardcover book ISBN 978-0-521-88068-8 is published by Cambridge University Press, http://www.cambridge.org. This electronic edition is published by Numerical Recipes Software, http://numerical.recipes. Permission is granted for authorized subscribers to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any public server computer, is strictly prohibited.