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

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

4.5 Gaussian Quadratures and Orthogonal Polynomials

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

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

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

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

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

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

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

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

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

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

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

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

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

```
float qgaus(float (*func)(float), float a, float b)
Returns the integral of the function func between a and b, by ten-point Gauss-Legendre inte-
gration: the function is evaluated exactly ten times at interior points in the range of integration.
{
    int j;
    float xr,xm,dx,s;
    static float x[]={0.0,0.1488743389,0.4333953941,     The abscissas and weights.
        0.6794095682,0.8650633666,0.9739065285};             First value of each array
    static float w[]={0.0,0.2955242247,0.2692667193,         not used.
        0.2190863625,0.1494513491,0.0666713443};

    xm=0.5*(b+a);
    xr=0.5*(b-a);
    s=0;                              Will be twice the average value of the function, since the
    for (j=1;j<=5;j++) {                  ten weights (five numbers above each used twice)
        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

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

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

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

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

where

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

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

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

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

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

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

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

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

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

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

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

Computation of the Abscissas and Weights

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

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

Gauss-Legendre:

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

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

Gauss-Chebyshev:

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

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

Gauss-Laguerre:

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

$$(j+1)L^\alpha_{j+1} = (-x+2j+\alpha+1)L^\alpha_j - (j+\alpha)L^\alpha_{j-1} \tag{4.5.12}$$

Gauss-Hermite:

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

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

Gauss-Jacobi:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The formula for the weights becomes

$$w_j = \frac{2}{(\widetilde{H}'_j)^2} \qquad (4.5.20)$$

while the formula for the derivative with this normalization is

$$\widetilde{H}'_j = \sqrt{2j}\,\widetilde{H}_{j-1} \qquad (4.5.21)$$

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

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

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

void gauher(float x[], float w[], int n)
```
Given n, this routine returns arrays x[1..n] and w[1..n] containing the abscissas and weights of the n-point Gauss-Hermite quadrature formula. The largest abscissa is returned in x[1], the most negative in x[n].
```
{
    void nrerror(char error_text[]);
    int i,its,j,m;
    double p1,p2,p3,pp,z,z1;                High precision is a good idea for this rou-
                                                tine.

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

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

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

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

void gaujac(float x[], float w[], int n, float alf, float bet)
```
Given alf and bet, the parameters α and β of the Jacobi polynomials, this routine returns arrays x[1..n] and w[1..n] containing the abscissas and weights of the n-point Gauss-Jacobi quadrature formula. The largest abscissa is returned in x[1], the smallest in x[n].
```
{
    float gammln(float xx);
    void nrerror(char error_text[]);
    int i,its,j;
    float alfbet,an,bn,r1,r2,r3;
    double a,b,c,p1,p2,p3,pp,temp,z,z1;     High precision is a good idea for this rou-
                                               tine.
    for (i=1;i<=n;i++) {                    Loop over the desired roots.
        if (i == 1) {                       Initial guess for the largest root.
            an=alf/n;
            bn=bet/n;
            r1=(1.0+alf)*(2.78/(4.0+n*n)+0.768*an/n);
            r2=1.0+1.48*an+0.96*bn+0.452*an*an+0.83*an*bn;
            z=1.0-r1/r2;
        } else if (i == 2) {                Initial guess for the second largest root.
            r1=(4.1+alf)/((1.0+alf)*(1.0+0.156*alf));
            r2=1.0+0.06*(n-8.0)*(1.0+0.12*alf)/n;
            r3=1.0+0.012*bet*(1.0+0.25*fabs(alf))/n;
            z -= (1.0-z)*r1*r2*r3;
        } else if (i == 3) {                Initial guess for the third largest root.
            r1=(1.67+0.28*alf)/(1.0+0.37*alf);
            r2=1.0+0.22*(n-8.0)/n;
            r3=1.0+8.0*bet/((6.28+bet)*n*n);
            z -= (x[1]-z)*r1*r2*r3;
        } else if (i == n-1) {              Initial guess for the second smallest root.
            r1=(1.0+0.235*bet)/(0.766+0.119*bet);
            r2=1.0/(1.0+0.639*(n-4.0)/(1.0+0.71*(n-4.0)));
            r3=1.0/(1.0+20.0*alf/((7.5+alf)*n*n));
            z += (z-x[n-3])*r1*r2*r3;
        } else if (i == n) {                Initial guess for the smallest root.
            r1=(1.0+0.37*bet)/(1.67+0.28*bet);
            r2=1.0/(1.0+0.22*(n-8.0)/n);
            r3=1.0/(1.0+8.0*alf/((6.28+alf)*n*n));
            z += (z-x[n-2])*r1*r2*r3;
        } else {                            Initial guess for the other roots.
            z=3.0*x[i-1]-3.0*x[i-2]+x[i-3];
        }
        alfbet=alf+bet;
        for (its=1;its<=MAXIT;its++) {      Refinement by Newton's method.
            temp=2.0+alfbet;                Start the recurrence with P0 and P1 to avoid
            p1=(alf-bet+temp*z)/2.0;           a division by zero when α + β = 0 or
            p2=1.0;                            −1.
            for (j=2;j<=n;j++) {            Loop up the recurrence relation to get the
                p3=p2;                         Jacobi polynomial evaluated at z.
                p2=p1;
                temp=2*j+alfbet;
                a=2*j*(j+alfbet)*(temp-2.0);
                b=(temp-1.0)*(alf*alf-bet*bet+temp*(temp-2.0)*z);
                c=2.0*(j-1+alf)*(j-1+bet)*temp;
                p1=(b*p2-c*p3)/a;
            }
            pp=(n*(alf-bet-temp*z)*p1+2.0*(n+alf)*(n+bet)*p2)/(temp*(1.0-z*z));
```
p1 is now the desired Jacobi polynomial. We next compute pp, its derivative, by a standard relation involving also p2, the polynomial of one lower order.
```
            z1=z;
            z=z1-p1/pp;                     Newton's formula.
            if (fabs(z-z1) <= EPS) break;
        }
        if (its > MAXIT) nrerror("too many iterations in gaujac");
        x[i]=z;                              Store the root and the weight.
        w[i]=exp(gammln(alf+n)+gammln(bet+n)-gammln(n+1.0)-
            gammln(n+alfbet+1.0))*temp*pow(2.0,alfbet)/(pp*p2);
    }
}
```

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

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

Case of Known Recurrences

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

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

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

or

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

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

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

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

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

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

where

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

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

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

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

    z=matrix(1,n,1,n);
    for (i=1;i<=n;i++) {
        if (i != 1) b[i]=sqrt(b[i]);        Set up superdiagonal of Jacobi matrix.
        for (j=1;j<=n;j++) z[i][j]=(float)(i == j);
        Set up identity matrix for tqli to compute eigenvectors.
    }
    tqli(a,b,n,z);
    eigsrt(a,z,n);                          Sort eigenvalues into descending order.
    for (i=1;i<=n;i++) {
        x[i]=a[i];
        w[i]=amu0*z[1][i]*z[1][i];          Equation (4.5.12).
    }
    free_matrix(z,1,n,1,n);
}
```

Orthogonal Polynomials with Nonclassical Weights

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

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

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

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

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

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

So assume that we know the *modified moments*

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

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

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

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

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

Initialize

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

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

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

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

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

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

Wheeler's algorithm requires that the modified moments (4.5.30) be accurately computed. In practical cases there is often a closed form, or else recurrence relations can be used. The algorithm is extremely successful for *finite* intervals (a, b). For infinite intervals, the algorithm does not completely remove the ill-conditioning. In this case, Gautschi [8,9] recommends reducing the interval to a finite interval by a change of variable, and then using a suitable discretization procedure to compute the inner products. You will have to consult the references for details.

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

```
#include "nrutil.h"

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

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

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

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

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

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

while the modified moments are

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

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

Extensions of Gaussian Quadrature

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

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

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

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.4. [1]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.6 Multidimensional Integrals

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

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

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

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

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