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

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

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

CITED REFERENCES AND FURTHER READING:

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

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

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

4.4 Improper Integrals

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

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

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

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

We need a workhorse like the extended trapezoidal rule (equation 4.1.11), but one which is an *open* formula in the sense of §4.1, i.e., does not require the integrand to be evaluated at the endpoints. Equation (4.1.19), the extended midpoint rule, is the best choice. The reason is that (4.1.19) shares with (4.1.11) the "deep" property of having an error series that is entirely even in h. Indeed there is a formula, not as well known as it ought to be, called the *Second Euler-Maclaurin summation formula*,

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

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

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

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

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

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

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

The routine midpnt can exactly replace trapzd in a driver routine like qtrap (§4.2); one simply changes trapzd(func,a,b,j) to midpnt(func,a,b, j), and perhaps also decreases the parameter JMAX since $3^{\mathtt{JMAX}-1}$ (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) substitutes midpnt for trapzd and decreases JMAX as above, but now also changes the extrapolation step to be

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

for a singularity at a, and

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

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

```
#include <math.h>

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

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

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

Similarly,

```
#include <math.h>

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

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

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

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

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

so that

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

The user-transparent implementation would be

```
#include <math.h>

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

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

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

CITED REFERENCES AND FURTHER READING:

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

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

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

4.5 Gaussian Quadratures and Orthogonal Polynomials

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

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

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

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

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

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

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

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

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