etemp = e;
e = d;
if (fabs(p) >= fabs(0.5*q*etemp) || p <= q*(a-x) || p >= q*(b-x))
d = CGOLD*(e = (x >= xm ? a-x : b-x));
The above conditions determine the acceptability of the parabolic fit. Here we take the golden section step into the larger of the two segments.
else {
    d = p/q;
u = x + d;
    if (u-a < tol2 || b-u < tol2)
        d = SIGN(tol1, x - x);
}
else {
    d = CGOLD*(e = (x >= xm ? a-x : b-x));
}
u = (fabs(d) >= tol1 ? x + d : x + SIGN(tol1, d));
This is the one function evaluation per iteration.
if (fu <= fx) {
    if (u >= x) a = x; else b = x;
    if (fu <= fw || w = x) {
        v = w;
u = w;
fw = fu;
    } else if (fu <= fv || v = x || v = w) {
        v = u;
    fv = fu;
}
}
else {
    if (u < x) a = u; else b = u;
    if (fu <= fv || v <= x) {
        v = w;
u = w;
fw = fu;
    } else if (fu <= fv || v <= x || v <= w) {
        v = u;
    fv = fu;
}
}  
nrerror("Too many iterations in brent");
*xmin = x;
return fx;

CITED REFERENCES AND FURTHER READING:

10.3 One-Dimensional Search with First Derivatives

Here we want to accomplish precisely the same goal as in the previous section, namely to isolate a functional minimum that is bracketed by the triplet of abscissas \((a, b, c)\), but utilizing an additional capability to compute the function’s first derivative as well as its value.
In principle, we might simply search for a zero of the derivative, ignoring the function value information, using a root finder like rtflsp or zbrent (§§9.2–9.3). It doesn’t take long to reject that idea: How do we distinguish maxima from minima? Where do we go from initial conditions where the derivatives on one or both of the outer bracketing points indicate that “downhill” is in the direction out of the bracketed interval?

We don’t want to give up our strategy of maintaining a rigorous bracket on the minimum at all times. The only way to keep such a bracket is to update it using function (not derivative) information, with the central point in the bracketing triplet always that with the lowest function value. Therefore the role of the derivatives can only be to help us choose new trial points within the bracket.

One school of thought is to “use everything you’ve got”: Compute a polynomial of relatively high order (cubic or above) that agrees with some number of previous function and derivative evaluations. For example, there is a unique cubic that agrees with function and derivative at two points, and one can jump to the interpolated minimum of that cubic (if there is a minimum within the bracket). Suggested by Davidon and others, formulas for this tactic are given in [1].

We like to be more conservative than this. Once superlinear convergence sets in, it hardly matters whether its order is moderately lower or higher. In practical problems that we have met, most function evaluations are spent in getting globally close enough to the minimum for superlinear convergence to commence. So we are more worried about all the funny “stiff” things that high-order polynomials can do (cf. Figure 3.0.1b), and about their sensitivities to roundoff error.

This leads us to use derivative information only as follows: The sign of the derivative at the central point of the bracketing triplet \((a, b, c)\) indicates uniquely whether the next test point should be taken in the interval \((a, b)\) or in the interval \((b, c)\). The value of this derivative and of the derivative at the second-best-so-far point are extrapolated to zero by the secant method (inverse linear interpolation), which by itself is superlinear of order 1.618. (The golden mean again: see [1], p. 57.) We impose the same sort of restrictions on this new trial point as in Brent’s method. If the trial point must be rejected, we bisect the interval under scrutiny.

Yes, we are fuddy-duddies when it comes to making flamboyant use of derivative information in one-dimensional minimization. But we have met too many functions whose computed “derivatives” don’t integrate up to the function value and don’t accurately point the way to the minimum, usually because of roundoff errors, sometimes because of truncation error in the method of derivative evaluation.

You will see that the following routine is closely modeled on brnt in the previous section.

```c
#include <math.h>
#include "nrutil.h"
#define ITMAX 100
#define ZEPS 1.0e-10
#define MOV3(a,b,c, d,e,f) (a)=(d);(b)=(e);(c)=(f);

float dbrent(float ax, float bx, float cx, float (*f)(float),
float (*df)(float), float tol, float *xmin)
Given a function \(f\) and its derivative function \(df\), and given a bracketing triplet of abscissas \(ax\), \(bx\), \(cx\) [such that \(bx\) is between \(ax\) and \(cx\), and \(f(bx)\) is less than both \(f(ax)\) and \(f(cx)\)], this routine isolates the minimum to a fractional precision of about \(tol\) using a modification of Brent’s method that uses derivatives. The abscissa of the minimum is returned as \(xmin\), and
```
the minimum function value is returned as `dbrent`, the returned function value.

```c
int iter, ok1, ok2;
float a, b, d1, d2, du, dv, dx, e = 0.0;
float fu, fv, fw, olde, tol1, tol2, u, u1, u2, v, w, x, xm;

Comments following will point out only differences from the routine `brent`. Read that routine first.

a = (ax < cx ? ax : cx);
b = (ax > cx ? ax : cx);
x = w = vb;
fv = fv = (*f)(x);
dv = dv = dx = (*df)(x);

for (iter=1; iter <= ITMAX; iter++) {
    xm = 0.5*(a + b);
tol1 = tol*fabs(x) + ZEPS;
tol2 = 2.0*tol1;
    if (fabs(x - xm) <= (tol2 - 0.5*(b - a))) {
        *xmin = x;
        return fx;
    }
    if (fabs(e) > tol1) {
        d1 = 2.0*(b - a);
        if (du == dx) d1 = (u - x)*dx / (dx - du);
        if (dv == dx) d2 = (v - x)*dx / (dx - dv);
        Which of these two estimates of d shall we take? We will insist that they be within
        the bracket, and on the side pointed to by the derivative at x:
        u1 = x + d1;
        u2 = x + d2;
        ok1 = (a - u1) * (u1 - b) > 0.0 && dx * d1 <= 0.0;
        ok2 = (a - u2) * (u2 - b) > 0.0 && dx * d2 <= 0.0;
        e = d;
        if (ok1 || ok2) {
            Take only an acceptable d, and if both are acceptable, then take
            the smallest one.
            if (ok1 && ok2)
                d = fabs(d1) < fabs(d2) ? d1 : d2;
            else if (ok1)
                d = d1;
            else
                d = d2;
        }
        if (fabs(d) <= fabs(0.5*olde)) {
            u = x + d;
            if (u - a <= tol2 || b - u <= tol2)
                d = SIGN(tol1, x - u);
        }
        else {
            Bisect, not golden section.
            d = 0.5*(e = (dx >= 0.0 ? a - x : b - x));
            Decide which segment by the sign of the derivative.
        }
    }
    else {
        d = 0.5*(e = (dx >= 0.0 ? a - x : b - x));
    }

    if (fabs(d) >= tol1) {
        u = x + d;
        fu = (*f)(u);
    }
    else {
        u = x + SIGN(tol1, d);
        fu = (*f)(u);
        if (fu > fx) {
            *xmin = x;
            return fx;
        }
    }
}
```

If the minimum step in the downhill direction takes us uphill, then we are done.
du=(*df)(u);
if (fu <= fx) {
    if (u >= x) a=x; else b=x;
    MOV3(v,fv,dv, u,fu,du)
    MOV3(w,fw,dw, x,fx,dx)
    MOV3(x,fx,dx, u,fu,du)
} else {
    if (u < x) a=u; else b=u;
    if (fu <= fw || w == x) {
        MOV3(v,fv,dv, w,fw,dw)
        MOV3(w,fw,dw, u,fu,du)
    } else if (fu < fv || v == x || v == w) {
        MOV3(v,fv,dv, u,fu,du)
    }
}
}
nrerror("Too many iterations in routine dbrent");
return 0.0;

CITED REFERENCES AND FURTHER READING:

10.4 Downhill Simplex Method in Multidimensions

With this section we begin consideration of multidimensional minimization, that is, finding the minimum of a function of more than one independent variable. This section stands apart from those which follow, however: All of the algorithms after this section will make explicit use of a one-dimensional minimization algorithm as a part of their computational strategy. This section implements an entirely self-contained strategy, in which one-dimensional minimization does not figure.

The downhill simplex method is due to Nelder and Mead [1]. The method requires only function evaluations, not derivatives. It is not very efficient in terms of the number of function evaluations that it requires. Powell’s method (§10.5) is almost surely faster in all likely applications. However, the downhill simplex method may frequently be the best method to use if the figure of merit is “get something working quickly” for a problem whose computational burden is small.

The method has a geometrical naturalness about it which makes it delightful to describe or work through:

A simplex is the geometrical figure consisting, in $N$ dimensions, of $N + 1$ points (or vertices) and all their interconnecting line segments, polygonal faces, etc. In two dimensions, a simplex is a triangle. In three dimensions it is a tetrahedron, not necessarily the regular tetrahedron. (The simplex method of linear programming,