```
        for (j=1;j<=MAXIT;j++) {
            xm=0.5*(xl+xh);
            fm=(*func)(xm);                     First of two function evaluations per it-
            s=sqrt(fm*fm-fl*fh);                    eration.
            if (s == 0.0) return ans;
            xnew=xm+(xm-xl)*((fl >= fh ? 1.0 : -1.0)*fm/s);     Updating formula.
            if (fabs(xnew-ans) <= xacc) return ans;
            ans=xnew;
            fnew=(*func)(ans);                  Second of two function evaluations per
            if (fnew == 0.0) return ans;            iteration.
            if (SIGN(fm,fnew) != fm) {          Bookkeeping to keep the root bracketed
                xl=xm;                              on next iteration.
                fl=fm;
                xh=ans;
                fh=fnew;
            } else if (SIGN(fl,fnew) != fl) {
                xh=ans;
                fh=fnew;
            } else if (SIGN(fh,fnew) != fh) {
                xl=ans;
                fl=fnew;
            } else nrerror("never get here.");
            if (fabs(xh-xl) <= xacc) return ans;
        }
        nrerror("zriddr exceed maximum iterations");
    }
    else {
        if (fl == 0.0) return x1;
        if (fh == 0.0) return x2;
        nrerror("root must be bracketed in zriddr.");
    }
    return 0.0;                                 Never get here.
}
```

CITED REFERENCES AND FURTHER READING:

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §8.3.

Ostrowski, A.M. 1966, *Solutions of Equations and Systems of Equations*, 2nd ed. (New York: Academic Press), Chapter 12.

Ridders, C.J.F. 1979, *IEEE Transactions on Circuits and Systems*, vol. CAS-26, pp. 979–980. [1]

9.3 Van Wijngaarden–Dekker–Brent Method

While secant and false position formally converge faster than bisection, one finds in practice pathological functions for which bisection converges more rapidly. These can be choppy, discontinuous functions, or even smooth functions if the second derivative changes sharply near the root. Bisection always halves the interval, while secant and false position can sometimes spend many cycles slowly pulling distant bounds closer to a root. Ridders' method does a much better job, but it too can sometimes be fooled. Is there a way to combine superlinear convergence with the sureness of bisection?

Yes. We can keep track of whether a supposedly superlinear method is actually converging the way it is supposed to, and, if it is not, we can intersperse bisection steps so as to guarantee *at least* linear convergence. This kind of super-strategy requires attention to bookkeeping detail, and also careful consideration of how roundoff errors can affect the guiding strategy. Also, we must be able to determine reliably when convergence has been achieved.

An excellent algorithm that pays close attention to these matters was developed in the 1960s by van Wijngaarden, Dekker, and others at the Mathematical Center in Amsterdam, and later improved by Brent [1]. For brevity, we refer to the final form of the algorithm as *Brent's method*. The method is *guaranteed* (by Brent) to converge, so long as the function can be evaluated within the initial interval known to contain a root.

Brent's method combines root bracketing, bisection, and *inverse quadratic interpolation* to converge from the neighborhood of a zero crossing. While the false position and secant methods assume approximately linear behavior between two prior root estimates, inverse quadratic interpolation uses three prior points to fit an inverse quadratic function (x as a quadratic function of y) whose value at $y = 0$ is taken as the next estimate of the root x. Of course one must have contingency plans for what to do if the root falls outside of the brackets. Brent's method takes care of all that. If the three point pairs are $[a, f(a)], [b, f(b)], [c, f(c)]$ then the interpolation formula (cf. equation 3.1.1) is

$$\begin{aligned} x = {} & \frac{[y - f(a)][y - f(b)]c}{[f(c) - f(a)][f(c) - f(b)]} + \frac{[y - f(b)][y - f(c)]a}{[f(a) - f(b)][f(a) - f(c)]} \\ & + \frac{[y - f(c)][y - f(a)]b}{[f(b) - f(c)][f(b) - f(a)]} \end{aligned} \tag{9.3.1}$$

Setting y to zero gives a result for the next root estimate, which can be written as

$$x = b + P/Q \tag{9.3.2}$$

where, in terms of

$$R \equiv f(b)/f(c), \qquad S \equiv f(b)/f(a), \qquad T \equiv f(a)/f(c) \tag{9.3.3}$$

we have

$$P = S\,[T(R - T)(c - b) - (1 - R)(b - a)] \tag{9.3.4}$$

$$Q = (T - 1)(R - 1)(S - 1) \tag{9.3.5}$$

In practice b is the current best estimate of the root and P/Q ought to be a "small" correction. Quadratic methods work well only when the function behaves smoothly; they run the serious risk of giving very bad estimates of the next root or causing machine failure by an inappropriate division by a very small number ($Q \approx 0$). Brent's method guards against this problem by maintaining brackets on the root and checking where the interpolation would land before carrying out the division. When the correction P/Q would not land within the bounds, or when the bounds are not collapsing rapidly enough, the algorithm takes a bisection step. Thus, Brent's method combines the sureness of bisection with the speed of a higher-order method when appropriate. We recommend it as the method of choice for general one-dimensional root finding where a function's values only (and not its derivative or functional form) are available.

```
#include <math.h>
#include "nrutil.h"
#define ITMAX 100                          Maximum allowed number of iterations.
#define EPS 3.0e-8                         Machine floating-point precision.

float zbrent(float (*func)(float), float x1, float x2, float tol)
```
Using Brent's method, find the root of a function `func` known to lie between `x1` and `x2`. The root, returned as `zbrent`, will be refined until its accuracy is `tol`.
```
{
    int iter;
    float a=x1,b=x2,c=x2,d,e,min1,min2;
    float fa=(*func)(a),fb=(*func)(b),fc,p,q,r,s,tol1,xm;

    if ((fa > 0.0 && fb > 0.0) || (fa < 0.0 && fb < 0.0))
        nrerror("Root must be bracketed in zbrent");
    fc=fb;
    for (iter=1;iter<=ITMAX;iter++) {
        if ((fb > 0.0 && fc > 0.0) || (fb < 0.0 && fc < 0.0)) {
            c=a;                           Rename a, b, c and adjust bounding interval
            fc=fa;                             d.
            e=d=b-a;
        }
        if (fabs(fc) < fabs(fb)) {
            a=b;
            b=c;
            c=a;
            fa=fb;
            fb=fc;
            fc=fa;
        }
        tol1=2.0*EPS*fabs(b)+0.5*tol;      Convergence check.
        xm=0.5*(c-b);
        if (fabs(xm) <= tol1 || fb == 0.0) return b;
        if (fabs(e) >= tol1 && fabs(fa) > fabs(fb)) {
            s=fb/fa;                       Attempt inverse quadratic interpolation.
            if (a == c) {
                p=2.0*xm*s;
                q=1.0-s;
            } else {
                q=fa/fc;
                r=fb/fc;
                p=s*(2.0*xm*q*(q-r)-(b-a)*(r-1.0));
                q=(q-1.0)*(r-1.0)*(s-1.0);
            }
            if (p > 0.0) q = -q;           Check whether in bounds.
            p=fabs(p);
            min1=3.0*xm*q-fabs(tol1*q);
            min2=fabs(e*q);
            if (2.0*p < (min1 < min2 ? min1 : min2)) {
                e=d;                       Accept interpolation.
                d=p/q;
            } else {
                d=xm;                      Interpolation failed, use bisection.
                e=d;
            }
        } else {                           Bounds decreasing too slowly, use bisection.
            d=xm;
            e=d;
        }
        a=b;                                Move last best guess to a.
        fa=fb;
        if (fabs(d) > tol1)                 Evaluate new trial root.
            b += d;
        else
            b += SIGN(tol1,xm);
        fb=(*func)(b);
    }
    nrerror("Maximum number of iterations exceeded in zbrent");
    return 0.0;                             Never get here.
}
```

CITED REFERENCES AND FURTHER READING:

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), Chapters 3, 4. [1]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §7.2.

9.4 Newton-Raphson Method Using Derivative

Perhaps the most celebrated of all one-dimensional root-finding routines is *Newton's method*, also called the *Newton-Raphson method*. This method is distinguished from the methods of previous sections by the fact that it requires the evaluation of both the function $f(x)$, *and* the derivative $f'(x)$, at arbitrary points x. The Newton-Raphson formula consists geometrically of extending the tangent line at a current point x_i until it crosses zero, then setting the next guess x_{i+1} to the abscissa of that zero-crossing (see Figure 9.4.1). Algebraically, the method derives from the familiar Taylor series expansion of a function in the neighborhood of a point,

$$f(x+\delta) \approx f(x) + f'(x)\delta + \frac{f''(x)}{2}\delta^2 + \dots. \qquad (9.4.1)$$

For small enough values of δ, and for well-behaved functions, the terms beyond linear are unimportant, hence $f(x+\delta) = 0$ implies

$$\delta = -\frac{f(x)}{f'(x)}. \qquad (9.4.2)$$

Newton-Raphson is not restricted to one dimension. The method readily generalizes to multiple dimensions, as we shall see in §9.6 and §9.7, below.

Far from a root, where the higher-order terms in the series *are* important, the Newton-Raphson formula can give grossly inaccurate, meaningless corrections. For instance, the initial guess for the root might be so far from the true root as to let the search interval include a local maximum or minimum of the function. This can be death to the method (see Figure 9.4.2). If an iteration places a trial guess near such a local extremum, so that the first derivative nearly vanishes, then Newton-Raphson sends its solution off to limbo, with vanishingly small hope of recovery.