```
    x0=ax;                                      At any given time we will keep track of four
    x3=cx;                                          points, x0,x1,x2,x3.
    if (fabs(cx-bx) > fabs(bx-ax)) {            Make x0 to x1 the smaller segment,
        x1=bx;
        x2=bx+C*(cx-bx);                        and fill in the new point to be tried.
    } else {
        x2=bx;
        x1=bx-C*(bx-ax);
    }
    f1=(*f)(x1);                                The initial function evaluations. Note that
    f2=(*f)(x2);                                    we never need to evaluate the function
    while (fabs(x3-x0) > tol*(fabs(x1)+fabs(x2))) {     at the original endpoints.
        if (f2 < f1) {                          One possible outcome,
            SHFT3(x0,x1,x2,R*x1+C*x3)           its housekeeping,
            SHFT2(f1,f2,(*f)(x2))               and a new function evaluation.
        } else {                                The other outcome,
            SHFT3(x3,x2,x1,R*x2+C*x0)
            SHFT2(f2,f1,(*f)(x1))               and its new function evaluation.
        }
    }                                           Back to see if we are done.
    if (f1 < f2) {                              We are done. Output the best of the two
        *xmin=x1;                                   current values.
        return f1;
    } else {
        *xmin=x2;
        return f2;
    }
}
```

10.2 Parabolic Interpolation and Brent's Method in One Dimension

We already tipped our hand about the desirability of parabolic interpolation in the previous section's `mnbrak` routine, but it is now time to be more explicit. A golden section search is designed to handle, in effect, the worst possible case of function minimization, with the uncooperative minimum hunted down and cornered like a scared rabbit. But why assume the worst? If the function is nicely parabolic near to the minimum — surely the generic case for sufficiently smooth functions — then the parabola fitted through any three points ought to take us in a single leap to the minimum, or at least very near to it (see Figure 10.2.1). Since we want to find an abscissa rather than an ordinate, the procedure is technically called *inverse parabolic interpolation*.

The formula for the abscissa x that is the minimum of a parabola through three points $f(a)$, $f(b)$, and $f(c)$ is

$$x = b - \frac{1}{2}\frac{(b-a)^2[f(b)-f(c)] - (b-c)^2[f(b)-f(a)]}{(b-a)[f(b)-f(c)] - (b-c)[f(b)-f(a)]} \qquad (10.2.1)$$

as you can easily derive. This formula fails only if the three points are collinear, in which case the denominator is zero (minimum of the parabola is infinitely far

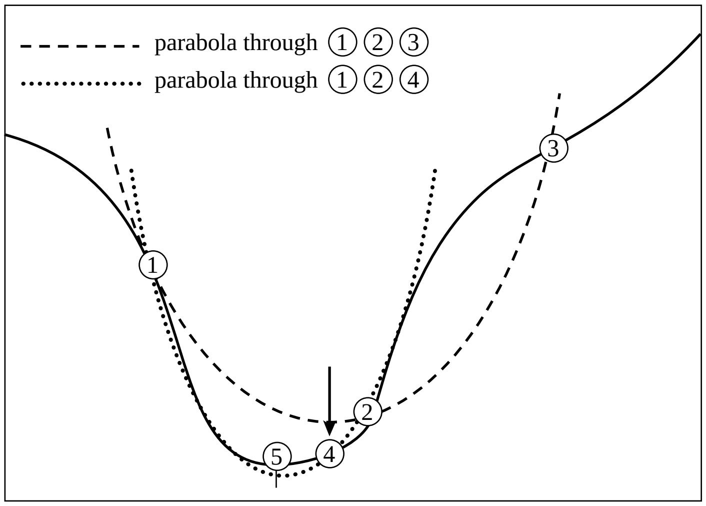

Figure 10.2.1. Convergence to a minimum by inverse parabolic interpolation. A parabola (dashed line) is drawn through the three original points 1,2,3 on the given function (solid line). The function is evaluated at the parabola's minimum, 4, which replaces point 3. A new parabola (dotted line) is drawn through points 1,4,2. The minimum of this parabola is at 5, which is close to the minimum of the function.

away). Note, however, that (10.2.1) is as happy jumping to a parabolic maximum as to a minimum. No minimization scheme that depends solely on (10.2.1) is likely to succeed in practice.

The exacting task is to invent a scheme that relies on a sure-but-slow technique, like golden section search, when the function is not cooperative, but that switches over to (10.2.1) when the function allows. The task is nontrivial for several reasons, including these: (i) The housekeeping needed to avoid unnecessary function evaluations in switching between the two methods can be complicated. (ii) Careful attention must be given to the "endgame," where the function is being evaluated very near to the roundoff limit of equation (10.1.2). (iii) The scheme for detecting a cooperative versus noncooperative function must be very robust.

Brent's method [1] is up to the task in all particulars. At any particular stage, it is keeping track of six function points (not necessarily all distinct), a, b, u, v, w and x, defined as follows: the minimum is bracketed between a and b; x is the point with the very least function value found so far (or the most recent one in case of a tie); w is the point with the second least function value; v is the previous value of w; u is the point at which the function was evaluated most recently. Also appearing in the algorithm is the point x_m, the midpoint between a and b; however, the function is not evaluated there.

You can read the code below to understand the method's logical organization. Mention of a few general principles here may, however, be helpful: Parabolic interpolation is attempted, fitting through the points x, v, and w. To be acceptable, the parabolic step must (i) fall within the bounding interval (a, b), and (ii) imply a movement from the best current value x that is *less* than half the movement of the *step before last*. This second criterion insures that the parabolic steps are actually converging to something, rather than, say, bouncing around in some nonconvergent limit cycle. In the worst possible case, where the parabolic steps are acceptable but useless, the method will approximately alternate between parabolic steps and golden sections, converging in due course by virtue of the latter. The reason for comparing to the step *before* last seems essentially heuristic: Experience shows that it is better not to "punish" the algorithm for a single bad step if it can make it up on the next one.

Another principle exemplified in the code is never to evaluate the function less than a distance `tol` from a point already evaluated (or from a known bracketing point). The reason is that, as we saw in equation (10.1.2), there is simply no information content in doing so: the function will differ from the value already evaluated only by an amount of order the roundoff error. Therefore in the code below you will find several tests and modifications of a potential new point, imposing this restriction. This restriction also interacts subtly with the test for "doneness," which the method takes into account.

A typical ending configuration for Brent's method is that a and b are $2 \times x \times \texttt{tol}$ apart, with x (the best abscissa) at the midpoint of a and b, and therefore fractionally accurate to $\pm\texttt{tol}$.

Indulge us a final reminder that `tol` should generally be no smaller than the square root of your machine's floating-point precision.

```
#include <math.h>
#include "nrutil.h"
#define ITMAX 100
#define CGOLD 0.3819660
#define ZEPS 1.0e-10
```
Here ITMAX is the maximum allowed number of iterations; CGOLD is the golden ratio; ZEPS is a small number that protects against trying to achieve fractional accuracy for a minimum that happens to be exactly zero.
```
#define SHFT(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);

float brent(float ax, float bx, float cx, float (*f)(float), float tol,
    float *xmin)
```
Given a function f, 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 Brent's method. The abscissa of the minimum is returned as xmin, and the minimum function value is returned as brent, the returned function value.
```
{
    int iter;
    float a,b,d,etemp,fu,fv,fw,fx,p,q,r,tol1,tol2,u,v,w,x,xm;
    float e=0.0;                                This will be the distance moved on
                                                     the step before last.
    a=(ax < cx ? ax : cx);                      a and b must be in ascending order,
    b=(ax > cx ? ax : cx);                          but input abscissas need not be.
    x=w=v=bx;                                   Initializations...
    fw=fv=fx=(*f)(x);
    for (iter=1;iter<=ITMAX;iter++) {           Main program loop.
        xm=0.5*(a+b);
        tol2=2.0*(tol1=tol*fabs(x)+ZEPS);
        if (fabs(x-xm) <= (tol2-0.5*(b-a))) {   Test for done here.
            *xmin=x;
            return fx;
        }
        if (fabs(e) > tol1) {                   Construct a trial parabolic fit.
            r=(x-w)*(fx-fv);
            q=(x-v)*(fx-fw);
            p=(x-v)*q-(x-w)*r;
            q=2.0*(q-r);
            if (q > 0.0) p = -p;
            q=fabs(q);
            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;                              Take the parabolic step.
                u=x+d;
                if (u-a < tol2 || b-u < tol2)
                    d=SIGN(tol1,xm-x);
            }
        } else {
            d=CGOLD*(e=(x >= xm ? a-x : b-x));
        }
        u=(fabs(d) >= tol1 ? x+d : x+SIGN(tol1,d));
        fu=(*f)(u);
        This is the one function evaluation per iteration.
        if (fu <= fx) {                             Now decide what to do with our func-
            if (u >= x) a=x; else b=x;                  tion evaluation.
            SHFT(v,w,x,u)                           Housekeeping follows:
            SHFT(fv,fw,fx,fu)
        } else {
            if (u < x) a=u; else b=u;
            if (fu <= fw || w == x) {
                v=w;
                w=u;
                fv=fw;
                fw=fu;
            } else if (fu <= fv || v == x || v == w) {
                v=u;
                fv=fu;
            }
        }                                           Done with housekeeping. Back for
    }                                                   another iteration.
    nrerror("Too many iterations in brent");
}
```

CITED REFERENCES AND FURTHER READING:

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 5. [1]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §8.2.

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.