```
    x3=cx
    if(abs(cx-bx).gt.abs(bx-ax))then        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)
    endif
    f1=f(x1)                        The initial function evaluations. Note that we never need to
    f2=f(x2)                            evaluate the function at the original endpoints.
1   if(abs(x3-x0).gt.tol*(abs(x1)+abs(x2)))then  Do-while loop: we keep returning here.
        if(f2.lt.f1)then            One possible outcome,
            x0=x1                   its housekeeping,
            x1=x2
            x2=R*x1+C*x3
            f1=f2
            f2=f(x2)                and a new function evaluation.
        else                        The other outcome,
            x3=x2
            x2=x1
            x1=R*x2+C*x0
            f2=f1
            f1=f(x1)                and its new function evaluation.
        endif
    goto 1                          Back to see if we are done.
    endif
    if(f1.lt.f2)then                We are done. Output the best of the two current values.
        golden=f1
        xmin=x1
    else
        golden=f2
        xmin=x2
    endif
    return
    END
```

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

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

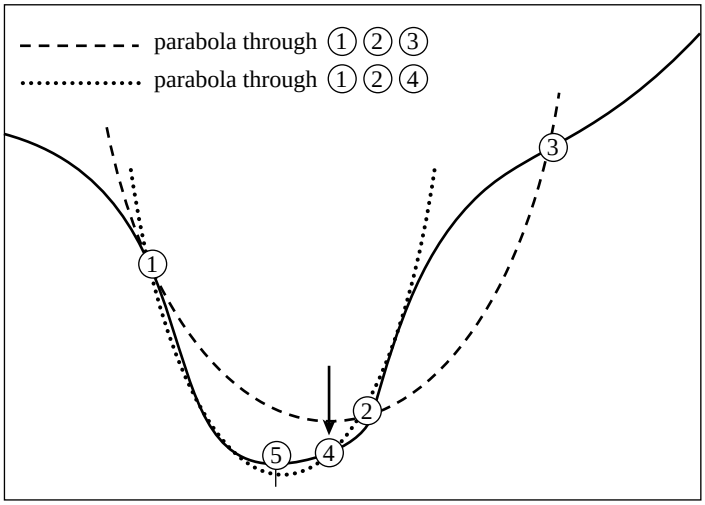

Figure 10.2.1. Convergence to a minimum by inverse parabolic interpolation. A parabola (dashed line) is drawn through the three original points 1,2,3 on the given function (solid line). The function is evaluated at the parabola's minimum, 4, which replaces point 3. A new parabola (dotted line) is drawn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

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 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 $2 \times x \times$ `tol` apart, with x (the best abscissa) at the midpoint of a and b, and therefore fractionally accurate to $\pm$`tol`.

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

```
FUNCTION brent(ax,bx,cx,f,tol,xmin)
INTEGER ITMAX
REAL brent,ax,bx,cx,tol,xmin,f,CGOLD,ZEPS
EXTERNAL f
PARAMETER (ITMAX=100,CGOLD=.3819660,ZEPS=1.0e-10)
    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.
    Parameters: Maximum allowed number of iterations; golden ratio; and a small number that
    protects against trying to achieve fractional accuracy for a minimum that happens to be
    exactly zero.
INTEGER iter
REAL a,b,d,e,etemp,fu,fv,fw,fx,p,q,r,tol1,tol2,u,v,w,x,xm
a=min(ax,cx)                        a and b must be in ascending order, though the input
b=max(ax,cx)                            abscissas need not be.
v=bx                                Initializations...
w=v
x=v
e=0.                                This will be the distance moved on the step before last.
fx=f(x)
fv=fx
fw=fx
do 11 iter=1,ITMAX                  Main program loop.
    xm=0.5*(a+b)
    tol1=tol*abs(x)+ZEPS
    tol2=2.*tol1
    if(abs(x-xm).le.(tol2-.5*(b-a))) goto 3      Test for done here.
    if(abs(e).gt.tol1) then                      Construct a trial parabolic fit.
        r=(x-w)*(fx-fv)
        q=(x-v)*(fx-fw)
        p=(x-v)*q-(x-w)*r
        q=2.*(q-r)
        if(q.gt.0.) p=-p
        q=abs(q)
        etemp=e
        e=d
```

```
        if(abs(p).ge.abs(.5*q*etemp).or.p.le.q*(a-x).or.
*            p.ge.q*(b-x)) goto 1
          The above conditions determine the acceptability of the parabolic fit. Here it is o.k.:
        d=p/q                       Take the parabolic step.
        u=x+d
        if(u-a.lt.tol2 .or. b-u.lt.tol2) d=sign(tol1,xm-x)
        goto 2                      Skip over the golden section step.
      endif
1     if(x.ge.xm) then              We arrive here for a golden section step, which we take
        e=a-x                         into the larger of the two segments.
      else
        e=b-x
      endif
      d=CGOLD*e                     Take the golden section step.
2     if(abs(d).ge.tol1) then       Arrive here with d computed either from parabolic fit, or
        u=x+d                         else from golden section.
      else
        u=x+sign(tol1,d)
      endif
      fu=f(u)                       This is the one function evaluation per iteration,
      if(fu.le.fx) then             and now we have to decide what to do with our function
        if(u.ge.x) then               evaluation. Housekeeping follows:
          a=x
        else
          b=x
        endif
        v=w
        fv=fw
        w=x
        fw=fx
        x=u
        fx=fu
      else
        if(u.lt.x) then
          a=u
        else
          b=u
        endif
        if(fu.le.fw .or. w.eq.x) then
          v=w
          fv=fw
          w=u
          fw=fu
        else if(fu.le.fv .or. v.eq.x .or. v.eq.w) then
          v=u
          fv=fu
        endif
      endif                         Done with housekeeping. Back for another iteration.
    enddo 11
    pause 'brent exceed maximum iterations'
3   xmin=x                          Arrive here ready to exit with best values.
    brent=fx
    return
    END
```

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.

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 `brent` in the previous section.