Numerical Recipes in C

The Art of Scientific Computing

William H. Press

Harvard-Smithsonian Center for Astrophysics

Brian P. Flannery

EXXON Research and Engineering Company

Saul A. Teukolsky

Department of Physics, Cornell University

William T. Vetterling

Polaroid Corporation

The right of the University of Cambridge to print and sell all manner of books was granted by Henry VIII in 1534. The University has printed and published continuously since 1584.

CAMBRIDGE UNIVERSITY PRESS
Cambridge
New York New Rochelle
Melbourne Sydney

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 which 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 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 which relies on a sure-but-slow technique, like golden section search, when the function is not cooperative, but which 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 (Brent, 1973) 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*

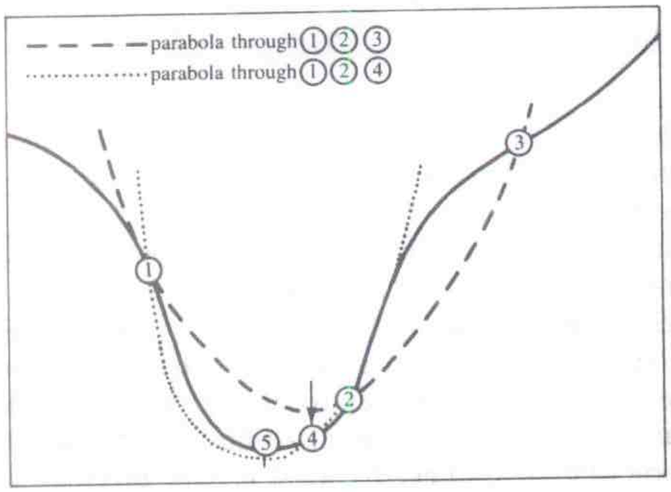

Figure 10.2.1. Convergence to a minimum by inverse parabolic interpolation. A parabola (dashed line) is drawn through the three original points 1,2,3 on the given function (solid line). The function is evaluated at the parabola's minimum, 4, which replaces point 3. A new parabola (dotted line) is drawn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

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.

```
#include <math.h>

#define ITMAX 100
#define CGOLD 0.3819660
#define ZEPS 1.0e-10
```
Maximum allowed number of iterations; golden ratio; and a small number which protects against trying to achieve fractional accuracy for a minimum that happens to be exactly zero.

```
#define SIGN(a,b) ((b)
#define SHFT(a,b,c,d)

float brent(ax,bx,cx,f
float ax,bx,cx,tol,*xm
float (*f)();      /* AN
```
Given a function f, and
between ax and cx, and f
minimum to a fractional
minimum is returned as
returned function value.
```
{
    int iter;
    float a,b,d,etemp,f
    float e=0.0;
    void nrerror();

    a=((ax < cx) ? ax :
    b=((ax > cx) ? ax :
    x=w=v=bx;
    fw=fv=fx=(*f)(x);
    for (iter=1;iter<=I
        xm=0.5*(a+b);
        tol2=2.0*(tol1=
        if (fabs(x-xm) <
            *xmin=x;
            return fx;
        }
        if (fabs(e) > to
            r=(x-w)*(fx-
            q=(x-v)*(fx-
            p=(x-v)*q-(x
            q=2.0*(q-r);
            if (q > 0.0)
            q=fabs(q);
            etemp=e;
            e=d;
            if (fabs(p) >
```
The above condit
section step into
```
                d=CGOLD*(
            else {
                d=p/q;
                u=x+d;
                if (u-a <
                    d=SIG
            }
        } else {
            d=CGOLD*(e=(x
        }
        u=(fabs(d) >= tol
        fu=(*f)(u);
        if (fu <= fx) {
            if (u >= x) a=
            SHFT(v,w,x,u)
            SHFT(fv,fw,fx,
        } else {
            if (u < x) a=u
            if (fu <= fw |
                v=w;
                w=u;
                fv=fw;
                fw=fu;
            } else if (fu
#define SIGN(a,b) ((b) > 0.0 ? fabs(a) : -fabs(a))
#define SHFT(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);

float brent(ax,bx,cx,f,tol,xmin)
float ax,bx,cx,tol,*xmin;
float (*f)();    /* ANSI: float (*f)(float); */
```

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
    void nrerror();                           last.

    a=((ax < cx) ? ax : cx);              a and b must be in ascending order, though the input
    b=((ax > cx) ? ax : cx);                  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;                      Arrive here ready to exit with best values.
            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))
            The above conditions determine the acceptability of the parabolic fit. Here we take the golden
            section step into the larger of the two segments.
                d=CGOLD*(e=(x >= xm ? a-x : b-x));
            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) {                   and now we have to decide what to do with our function
            if (u >= x) a=x; else b=x;        evaluation. Housekeeping follows:
            SHFT(v,w,x,u)
            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");
  *xmin=x;
  return fx;
}
```

REFERENCES AND FURTHER READING:

Brent, Richard P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, N.J.: Prentice-Hall), Chapter 5.

Forsythe, George E., Malcolm, Michael A., and Moler, Cleve B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, N.J.: 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). 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) which 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 Acton.

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 tl
getting globally close
commence. So we are
order polynomials ca
roundoff error.

This leads us to
the derivative at the c
uniquely whether the
or in the interval $(b, c$
the second-best-so-far
(inverse linear interp
(The golden mean ag
restrictions on this ne
must be rejected, we

Yes, we are fuddy
derivative information
bellyful of functions w
function value and *don*
cause of roundoff error
of derivative evaluatio

You will see that
the previous section.

```
#include <math.h>

#define ITMAX 100
#define ZEPS 1.0e-10
#define SIGN(a,b) ((b) >
#define MOV3(a,b,c, d,e,

float dbrent(ax,bx,cx,f,
float ax,bx,cx,tol,*xmin
float (*f)(),(*df)(); /*
Given a function f and its d
ax, bx, cx [such that bx is be
this routine isolates the mini
of Brent's method that uses
and the minimum function va
{
    int iter,ok1,ok2;
    float a,b,d,d1,d2,du,
    float fu,fv,fw,fx,old
    void nrerror();

    Comments following will poin
    a=(ax < cx ? ax : cx);
    b=(ax > cx ? ax : cx);
    x=w=v=bx;
    fw=fv=fx=(*f)(x);
    dw=dv=dx=(*df)(x);
    for (iter=1;iter<=ITMA
        xm=0.5*(a+b);
        tol1=tol*fabs(x)+Z
        tol2=2.0*tol1;
        if (fabs(x-xm) <= (
```

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 Acton, 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 had a bellyful of 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.

```
#include <math.h>

#define ITMAX 100
#define ZEPS 1.0e-10
#define SIGN(a,b) ((b) > 0.0 ? fabs(a) : -fabs(a))
#define MOV3(a,b,c, d,e,f) (a)=(d);(b)=(e);(c)=(f);

float dbrent(ax,bx,cx,f,df,tol,xmin)
float ax,bx,cx,tol,*xmin;
float (*f)(),(*df)(); /* ANSI: float (*f)(float),(*df)(float); */
```

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.

```
{
    int iter,ok1,ok2;             The oks will be used as flags for whether proposed steps are
    float a,b,d,d1,d2,du,dv,dw,dx,e=0.0;     acceptable or not.
    float fu,fv,fw,fx,olde,tol1,tol2,u,u1,u2,v,w,x,xm;
    void nrerror();

    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=v=bx;
    fw=fv=fx=(*f)(x);
    dw=dv=dx=(*df)(x);            All our housekeeping chores are doubled by the necessity of mov-
    for (iter=1;iter<=ITMAX;iter++) {       ing derivative values around as well as function
        xm=0.5*(a+b);                        values.
        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);              Initialize these d's to an out-of-bracket value.
            d2=d1;
            if (dw != dx) d1=(w-x)*dx/(dx-dw);        Secant method, first on one, then on
            if (dv != dx) d2=(v-x)*dx/(dx-dv);        the other, point.
            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;
            olde=e;                    Movement on the step before last.
            e=d;
            if (ok1 || ok2) {          Take only an acceptable d, and if both are acceptable, then
                if (ok1 && ok2)            take the smallest one.
                    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,xm-x);
                } else {
                    d=0.5*(e=(dx >= 0.0 ? a-x : b-x));       Bisect, not golden section.
                }                      Decide which segment by the sign of the derivative.
            } else {
                d=0.5*(e=(dx >= 0.0 ? a-x : b-x));
            }
        } 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) {             If the minimum step in the downhill direction takes us uphill, then
                *xmin=x;                   we are done.
                return fx;
            }
        }
        du=(*df)(u);                   Now all the housekeeping, sigh.
        if (fu <= fx) {
            if (u >= x) a=x; else b=x;
            MOV3(v,fv,dv, w,fw,dw)
            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");
}
```

REFERENCES AND

Acton, Form
Harper

Brent, Richa
(Englev

10.4 Downhil Multidir

With this sectio
tion, that is, finding t
variable. This sectio
the algorithms after t
minimization algorith
tional strategy. This
in which one-dimensi

The *downhill si*
method requires only
efficient in terms of th
ell's method (§10.5) is
the downhill simplex
the figure of merit is
computational burde

The method has
lightful to describe o

A *simplex* is the
1 points (or vertices)
faces, etc. In two din
it is a tetrahedron, n
method of linear prog
a simplex. Otherwise i
describing in this secti
are nondegenerate, i.e.
any point of a nondege
points define vector di

In one-dimensiona
so that the success of
is no analogous proce
minimization, the best
an N-vector of indepe
is then supposed to m

value.

:hod, first on one, then on
point.
: that they be within the

both are acceptable, then

3isect, not golden section.
the derivative.

ection takes us uphill, then

REFERENCES AND FURTHER READING:

Acton, Forman S. 1970, *Numerical Methods That Work* (New York: Harper and Row), p. 55, pp. 454–458.

Brent, Richard P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, N.J.: Prentice-Hall), p. 78.

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 the one-dimensional minimization algorithms of §10.1, §10.2, or §10.3 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 (1965). 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 also makes use of the geometrical concept of a simplex. Otherwise it is completely unrelated to the algorithm that we are describing in this section.) In general we are only interested in simplexes that are nondegenerate, i.e. which enclose a finite inner N-dimensional volume. If any point of a nondegenerate simplex is taken as the origin, then the N other points define vector directions that span the N-dimensional vector space.

In one-dimensional minimization, it was possible to bracket a minimum, so that the success of a subsequent isolation was guaranteed. Alas! There is no analogous procedure in multidimensional space. For multidimensional minimization, the best we can do is give our algorithm a starting guess, that is, an N-vector of independent variables as the first point to try. The algorithm is then supposed to make its own way downhill through the unimaginable