```
            }
        }
        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");
    return 0.0;                                       Never get here.
}
```

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 55; 454–458. [1]

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: 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 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,

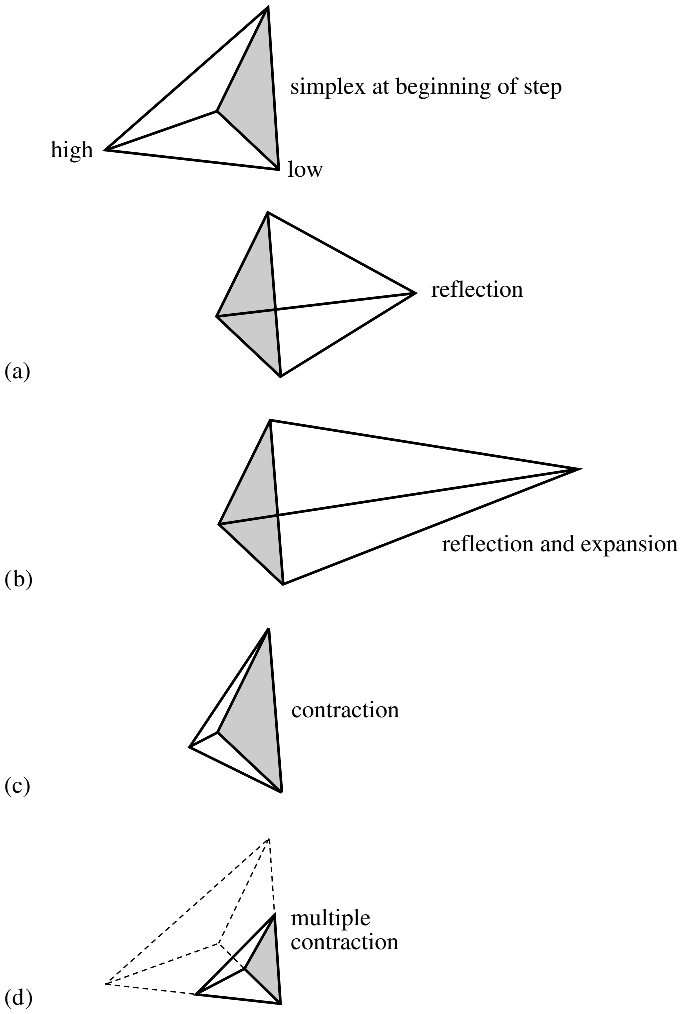

Figure 10.4.1. Possible outcomes for a step in the downhill simplex method. The simplex at the beginning of the step, here a tetrahedron, is shown, top. The simplex at the end of the step can be any one of (a) a reflection away from the high point, (b) a reflection and expansion away from the high point, (c) a contraction along one dimension from the high point, or (d) a contraction along all dimensions towards the low point. An appropriate sequence of such steps will always converge to a minimum of the function.

described in §10.8, 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., that 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 complexity of an N-dimensional topography, until it encounters a (local, at least) minimum.

The downhill simplex method must be started not just with a single point, but with $N + 1$ points, defining an initial simplex. If you think of one of these points (it matters not which) as being your initial starting point $\mathbf{P}_0$, then you can take the other N points to be

$$\mathbf{P}_i = \mathbf{P}_0 + \lambda \mathbf{e}_i \tag{10.4.1}$$

where the $\mathbf{e}_i$'s are N unit vectors, and where λ is a constant which is your guess of the problem's characteristic length scale. (Or, you could have different λ_i's for each vector direction.)

The downhill simplex method now takes a series of steps, most steps just moving the point of the simplex where the function is largest ("highest point") through the opposite face of the simplex to a lower point. These steps are called reflections, and they are constructed to conserve the volume of the simplex (hence maintain its nondegeneracy). When it can do so, the method expands the simplex in one or another direction to take larger steps. When it reaches a "valley floor," the method contracts itself in the transverse direction and tries to ooze down the valley. If there is a situation where the simplex is trying to "pass through the eye of a needle," it contracts itself in all directions, pulling itself in around its lowest (best) point. The routine name `amoeba` is intended to be descriptive of this kind of behavior; the basic moves are summarized in Figure 10.4.1.

Termination criteria can be delicate in any multidimensional minimization routine. Without bracketing, and with more than one independent variable, we no longer have the option of requiring a certain tolerance for a single independent variable. We typically can identify one "cycle" or "step" of our multidimensional algorithm. It is then possible to terminate when the vector distance moved in that step is fractionally smaller in magnitude than some tolerance `tol`. Alternatively, we could require that the decrease in the function value in the terminating step be fractionally smaller than some tolerance `ftol`. Note that while `tol` should not usually be smaller than the square root of the machine precision, it is perfectly appropriate to let `ftol` be of order the machine precision (or perhaps slightly larger so as not to be diddled by roundoff).

Note well that either of the above criteria might be fooled by a single anomalous step that, for one reason or another, failed to get anywhere. Therefore, it is frequently a good idea to *restart* a multidimensional minimization routine at a point where it claims to have found a minimum. For this restart, you should reinitialize any ancillary input quantities. In the downhill simplex method, for example, you should reinitialize N of the $N + 1$ vertices of the simplex again by equation (10.4.1), with $\mathbf{P}_0$ being one of the vertices of the claimed minimum.

Restarts should never be very expensive; your algorithm did, after all, converge to the restart point once, and now you are starting the algorithm already there.

Consider, then, our N-dimensional amoeba:

```
#include <math.h>
#include "nrutil.h"
#define TINY 1.0e-10                    A small number.
#define NMAX 5000                       Maximum allowed number of function evalua-
#define GET_PSUM \                          tions.
                    for (j=1;j<=ndim;j++) {\
                    for (sum=0.0,i=1;i<=mpts;i++) sum += p[i][j];\
                    psum[j]=sum;}
#define SWAP(a,b) {swap=(a);(a)=(b);(b)=swap;}

void amoeba(float **p, float y[], int ndim, float ftol,
    float (*funk)(float []), int *nfunk)
```

Multidimensional minimization of the function `funk(x)` where `x[1..ndim]` is a vector in `ndim` dimensions, by the downhill simplex method of Nelder and Mead. The matrix `p[1..ndim+1][1..ndim]` is input. Its `ndim+1` rows are `ndim`-dimensional vectors which are the vertices of the starting simplex. Also input is the vector `y[1..ndim+1]`, whose components must be pre-initialized to the values of `funk` evaluated at the `ndim+1` vertices (rows) of `p`; and `ftol` the fractional convergence tolerance to be achieved in the function value (n.b.!). On output, `p` and `y` will have been reset to `ndim+1` new points all within `ftol` of a minimum function value, and `nfunk` gives the number of function evaluations taken.

```
{
    float amotry(float **p, float y[], float psum[], int ndim,
        float (*funk)(float []), int ihi, float fac);
    int i,ihi,ilo,inhi,j,mpts=ndim+1;
    float rtol,sum,swap,ysave,ytry,*psum;

    psum=vector(1,ndim);
    *nfunk=0;
    GET_PSUM
    for (;;) {
        ilo=1;
        First we must determine which point is the highest (worst), next-highest, and lowest
        (best), by looping over the points in the simplex.
        ihi = y[1]>y[2] ? (inhi=2,1) : (inhi=1,2);
        for (i=1;i<=mpts;i++) {
            if (y[i] <= y[ilo]) ilo=i;
            if (y[i] > y[ihi]) {
                inhi=ihi;
                ihi=i;
            } else if (y[i] > y[inhi] && i != ihi) inhi=i;
        }
        rtol=2.0*fabs(y[ihi]-y[ilo])/(fabs(y[ihi])+fabs(y[ilo])+TINY);
        Compute the fractional range from highest to lowest and return if satisfactory.
        if (rtol < ftol) {              If returning, put best point and value in slot 1.
            SWAP(y[1],y[ilo])
            for (i=1;i<=ndim;i++) SWAP(p[1][i],p[ilo][i])
            break;
        }
        if (*nfunk >= NMAX) nrerror("NMAX exceeded");
        *nfunk += 2;
        Begin a new iteration. First extrapolate by a factor -1 through the face of the simplex
        across from the high point, i.e., reflect the simplex from the high point.
        ytry=amotry(p,y,psum,ndim,funk,ihi,-1.0);
        if (ytry <= y[ilo])
            Gives a result better than the best point, so try an additional extrapolation by a
            factor 2.
            ytry=amotry(p,y,psum,ndim,funk,ihi,2.0);
        else if (ytry >= y[inhi]) {
            The reflected point is worse than the second-highest, so look for an intermediate
            lower point, i.e., do a one-dimensional contraction.
            ysave=y[ihi];
            ytry=amotry(p,y,psum,ndim,funk,ihi,0.5);
            if (ytry >= ysave) {        Can't seem to get rid of that high point. Better
                for (i=1;i<=mpts;i++) {     contract around the lowest (best) point.
```

```
                if (i != ilo) {
                    for (j=1;j<=ndim;j++)
                        p[i][j]=psum[j]=0.5*(p[i][j]+p[ilo][j]);
                    y[i]=(*funk)(psum);
                }
            }
            *nfunk += ndim;         Keep track of function evaluations.
            GET_PSUM                Recompute psum.
        }
    } else --(*nfunk);              Correct the evaluation count.
}                                   Go back for the test of doneness and the next
free_vector(psum,1,ndim);               iteration.
}
```

```
#include "nrutil.h"

float amotry(float **p, float y[], float psum[], int ndim,
    float (*funk)(float []), int ihi, float fac)
```

Extrapolates by a factor `fac` through the face of the simplex across from the high point, tries it, and replaces the high point if the new point is better.

```
{
    int j;
    float fac1,fac2,ytry,*ptry;

    ptry=vector(1,ndim);
    fac1=(1.0-fac)/ndim;
    fac2=fac1-fac;
    for (j=1;j<=ndim;j++) ptry[j]=psum[j]*fac1-p[ihi][j]*fac2;
    ytry=(*funk)(ptry);          Evaluate the function at the trial point.
    if (ytry < y[ihi]) {         If it's better than the highest, then replace the highest.
        y[ihi]=ytry;
        for (j=1;j<=ndim;j++) {
            psum[j] += ptry[j]-p[ihi][j];
            p[ihi][j]=ptry[j];
        }
    }
    free_vector(ptry,1,ndim);
    return ytry;
}
```

CITED REFERENCES AND FURTHER READING:

Nelder, J.A., and Mead, R. 1965, *Computer Journal*, vol. 7, pp. 308–313. [1]

Yarbro, L.A., and Deming, S.N. 1974, *Analytica Chimica Acta*, vol. 73, pp. 391–398.

Jacoby, S.L.S, Kowalik, J.S., and Pizzo, J.T. 1972, *Iterative Methods for Nonlinear Optimization Problems* (Englewood Cliffs, NJ: Prentice-Hall).

10.5 Direction Set (Powell's) Methods in Multidimensions

We know (§10.1–§10.3) how to minimize a function of one variable. If we start at a point $\mathbf{P}$ in N-dimensional space, and proceed from there in some vector