Chapter 10. Minimization or Maximization of Functions

10.0 Introduction

In a nutshell: You are given a single function f that depends on one or more independent variables. You want to find the value of those variables where f takes on a maximum or a minimum value. You can then calculate what value of f is achieved at the maximum or minimum. The tasks of maximization and minimization are trivially related to each other, since one person's function f could just as well be another's $-f$. The computational desiderata are the usual ones: Do it quickly, cheaply, and in small memory. Often the computational effort is dominated by the cost of evaluating f (and also perhaps its partial derivatives with respect to all variables, if the chosen algorithm requires them). In such cases the desiderata are sometimes replaced by the simple surrogate: Evaluate f as few times as possible.

An extremum (maximum or minimum point) can be either *global* (truly the highest or lowest function value) or *local* (the highest or lowest in a finite neighborhood and not on the boundary of that neighborhood). (See Figure 10.0.1.) Finding a global extremum is, in general, a very difficult problem. Two standard heuristics are widely used: (i) find local extrema starting from widely varying starting values of the independent variables (perhaps chosen quasi-randomly, as in §7.7), and then pick the most extreme of these (if they are not all the same); or (ii) perturb a local extremum by taking a finite amplitude step away from it, and then see if your routine returns you to a better point, or "always" to the same one. Relatively recently, so-called "simulated annealing methods" (§10.9) have demonstrated important successes on a variety of global extremization problems.

Our chapter title could just as well be *optimization*, which is the usual name for this very large field of numerical research. The importance ascribed to the various tasks in this field depends strongly on the particular interests of whom you talk to. Economists, and some engineers, are particularly concerned with *constrained optimization*, where there are *a priori* limitations on the allowed values of independent variables. For example, the production of wheat in the U.S. must be a nonnegative number. One particularly well-developed area of constrained optimization is *linear programming*, where both the function to be optimized and the constraints happen to be linear functions of the independent variables. Section 10.8, which is otherwise somewhat disconnected from the rest of the material that we have chosen to include in this chapter, implements the so-called "simplex algorithm" for linear programming problems.

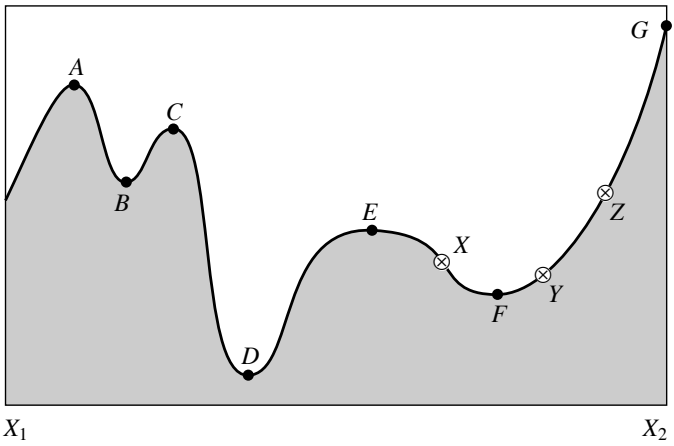


Figure 10.0.1. Extrema of a function in an interval. Points A, C, and E are local, but not global maxima. Points B and F are local, but not global minima. The global maximum occurs at G, which is on the boundary of the interval so that the derivative of the function need not vanish there. The global minimum is at D. At point E, derivatives higher than the first vanish, a situation which can cause difficulty for some algorithms. The points X, Y, and Z are said to "bracket" the minimum F, since Y is less than both X and Z.

One other section, §10.9, also lies outside of our main thrust, but for a different reason: so-called "annealing methods" are relatively new, so we do not yet know where they will ultimately fit into the scheme of things. However, these methods have solved some problems previously thought to be practically insoluble; they address directly the problem of finding global extrema in the presence of large numbers of undesired local extrema.

The other sections in this chapter constitute a selection of the best established algorithms in unconstrained minimization. (For definiteness, we will henceforth regard the optimization problem as that of minimization.) These sections are connected, with later ones depending on earlier ones. If you are just looking for the one "perfect" algorithm to solve your particular application, you may feel that we are telling you more than you want to know. Unfortunately, there is *no* perfect optimization algorithm. This is a case where we strongly urge you to try more than one method in comparative fashion. Your initial choice of method can be based on the following considerations:

- You must choose between methods that need only evaluations of the function to be minimized and methods that also require evaluations of the derivative of that function. In the multidimensional case, this derivative is the gradient, a vector quantity. Algorithms using the derivative are somewhat more powerful than those using only the function, but not always enough so as to compensate for the additional calculations of derivatives. We can easily construct examples favoring one approach or favoring the other. However, if you *can* compute derivatives, be prepared to try using them.
- For one-dimensional minimization (minimize a function of one variable) *without* calculation of the derivative, bracket the minimum as described in §10.1, and then use *Brent's method* as described in §10.2. If your function has a discontinuous second (or lower) derivative, then the parabolic

interpolations of Brent's method are of no advantage, and you might wish to use the simplest form of *golden section search*, as described in §10.1.
- For one-dimensional minimization *with* calculation of the derivative, §10.3 supplies a variant of Brent's method which makes limited use of the first derivative information. We shy away from the alternative of using derivative information to construct high-order interpolating polynomials. In our experience the improvement in convergence very near a smooth, analytic minimum does not make up for the tendency of polynomials sometimes to give wildly wrong interpolations at early stages, especially for functions that may have sharp, "exponential" features.

We now turn to the multidimensional case, both with and without computation of first derivatives.

- You must choose between methods that require storage of order N^2 and those that require only of order N, where N is the number of dimensions. For moderate values of N and reasonable memory sizes this is not a serious constraint. There will be, however, the occasional application where storage may be critical.
- We give in §10.4 a sometimes overlooked *downhill simplex method* due to Nelder and Mead. (This use of the word "simplex" is not to be confused with the simplex method of linear programming.) This method just crawls downhill in a straightforward fashion that makes almost no special assumptions about your function. This can be extremely slow, but it can also, in some cases, be extremely robust. Not to be overlooked is the fact that the code is concise and completely self-contained: a general N-dimensional minimization program in under 100 program lines! This method is most useful when the minimization calculation is only an incidental part of your overall problem. The storage requirement is of order N^2, and derivative calculations are not required.
- Section 10.5 deals with *direction-set methods*, of which *Powell's method* is the prototype. These are the methods of choice when you cannot easily calculate derivatives, and are not necessarily to be sneered at even if you can. Although derivatives are not needed, the method does require a one-dimensional minimization sub-algorithm such as Brent's method (see above). Storage is of order N^2.

There are two major families of algorithms for multidimensional minimization *with* calculation of first derivatives. Both families require a one-dimensional minimization sub-algorithm, which can itself either use, or not use, the derivative information, as you see fit (depending on the relative effort of computing the function and of its gradient vector). We do not think that either family dominates the other in all applications; you should think of them as available alternatives:

- The first family goes under the name *conjugate gradient methods*, as typified by the *Fletcher-Reeves algorithm* and the closely related and probably superior *Polak-Ribiere algorithm*. Conjugate gradient methods require only of order a few times N storage, require derivative calculations and

one-dimensional sub-minimization. Turn to §10.6 for detailed discussion and implementation.

- The second family goes under the names *quasi-Newton* or *variable metric* methods, as typified by the *Davidon-Fletcher-Powell (DFP)* algorithm (sometimes referred to just as *Fletcher-Powell*) or the closely related *Broyden-Fletcher-Goldfarb-Shanno (BFGS)* algorithm. These methods require of order N^2 storage, require derivative calculations and one-dimensional sub-minimization. Details are in §10.7.

You are now ready to proceed with scaling the peaks (and/or plumbing the depths) of practical optimization.

CITED REFERENCES AND FURTHER READING:

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall).

Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press).

Gill, P.E., Murray, W., and Wright, M.H. 1981, *Practical Optimization* (New York: Academic Press).

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 17.

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.1.

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

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 10.

10.1 Golden Section Search in One Dimension

Recall how the bisection method finds roots of functions in one dimension (§9.1): The root is supposed to have been bracketed in an interval (a, b). One then evaluates the function at an intermediate point x and obtains a new, smaller bracketing interval, either (a, x) or (x, b). The process continues until the bracketing interval is acceptably small. It is optimal to choose x to be the midpoint of (a, b) so that the decrease in the interval length is maximized when the function is as uncooperative as it can be, i.e., when the luck of the draw forces you to take the bigger bisected segment.

There is a precise, though slightly subtle, translation of these considerations to the minimization problem: What does it mean to *bracket* a minimum? A root of a function is known to be bracketed by a pair of points, a and b, when the function has opposite sign at those two points. A minimum, by contrast, is known to be bracketed only when there is a *triplet* of points, $a < b < c$ (or $c < b < a$), such that $f(b)$ is less than both $f(a)$ and $f(c)$. In this case we know that the function (if it is nonsingular) has a minimum in the interval (a, c).

The analog of bisection is to choose a new point x, either between a and b or between b and c. Suppose, to be specific, that we make the latter choice. Then we evaluate $f(x)$. If $f(b) < f(x)$, then the new bracketing triplet of points is (a, b, x);

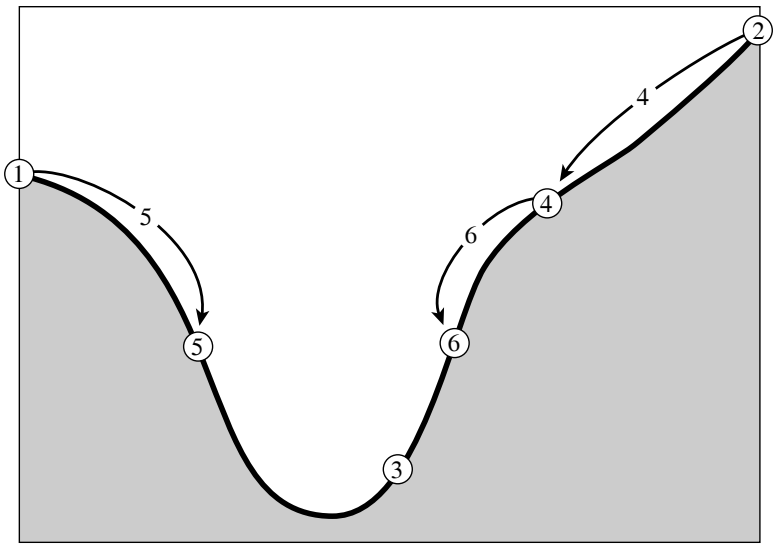


Figure 10.1.1. Successive bracketing of a minimum. The minimum is originally bracketed by points 1,3,2. The function is evaluated at 4, which replaces 2; then at 5, which replaces 1; then at 6, which replaces 4. The rule at each stage is to keep a center point that is lower than the two outside points. After the steps shown, the minimum is bracketed by points 5,3,6.

contrariwise, if $f(b) > f(x)$, then the new bracketing triplet is (b, x, c). In all cases the middle point of the new triplet is the abscissa whose ordinate is the best minimum achieved so far; see Figure 10.1.1. We continue the process of bracketing until the distance between the two outer points of the triplet is tolerably small.

How small is "tolerably" small? For a minimum located at a value b, you might naively think that you will be able to bracket it in as small a range as $(1 - \epsilon)b < b < (1 + \epsilon)b$, where ϵ is your computer's floating-point precision, a number like 3×10^{-8} (for `float`) or 10^{-15} (for `double`). Not so! In general, the shape of your function $f(x)$ near b will be given by Taylor's theorem

$$f(x) \approx f(b) + \frac{1}{2} f''(b)(x - b)^2 \tag{10.1.1}$$

The second term will be negligible compared to the first (that is, will be a factor ϵ smaller and will act just like zero when added to it) whenever

$$|x - b| < \sqrt{\epsilon}|b| \sqrt{\frac{2\,|f(b)|}{b^2 f''(b)}} \tag{10.1.2}$$

The reason for writing the right-hand side in this way is that, for most functions, the final square root is a number of order unity. Therefore, as a rule of thumb, it is hopeless to ask for a bracketing interval of width less than $\sqrt{\epsilon}$ times its central value, a fractional width of only about 10^{-4} (single precision) or 3×10^{-8} (double precision). Knowing this inescapable fact will save you a lot of useless bisections!

The minimum-finding routines of this chapter will often call for a user-supplied argument `tol`, and return with an abscissa whose fractional precision is about $\pm$`tol` (bracketing interval of fractional size about $2\times$`tol`). Unless you have a better

estimate for the right-hand side of equation (10.1.2), you should set `tol` equal to (not much less than) the square root of your machine's floating-point precision, since smaller values will gain you nothing.

It remains to decide on a strategy for choosing the new point x, given (a, b, c). Suppose that b is a fraction w of the way between a and c, i.e.

$$\frac{b-a}{c-a} = w \qquad \frac{c-b}{c-a} = 1-w \tag{10.1.3}$$

Also suppose that our next trial point x is an additional fraction z beyond b,

$$\frac{x-b}{c-a} = z \tag{10.1.4}$$

Then the next bracketing segment will either be of length $w+z$ relative to the current one, or else of length $1-w$. If we want to minimize the worst case possibility, then we will choose z to make these equal, namely

$$z = 1-2w \tag{10.1.5}$$

We see at once that the new point is the symmetric point to b in the original interval, namely with $|b-a|$ equal to $|x-c|$. This implies that the point x lies in the larger of the two segments (z is positive only if $w < 1/2$).

But where in the larger segment? Where did the value of w itself come from? Presumably from the previous stage of applying our same strategy. Therefore, if z is chosen to be optimal, then so was w before it. This *scale similarity* implies that x should be the same fraction of the way from b to c (if that is the bigger segment) as was b from a to c, in other words,

$$\frac{z}{1-w} = w \tag{10.1.6}$$

Equations (10.1.5) and (10.1.6) give the quadratic equation

$$w^2 - 3w + 1 = 0 \qquad \text{yielding} \qquad w = \frac{3-\sqrt{5}}{2} \approx 0.38197 \tag{10.1.7}$$

In other words, the optimal bracketing interval (a, b, c) has its middle point b a fractional distance 0.38197 from one end (say, a), and 0.61803 from the other end (say, b). These fractions are those of the so-called *golden mean* or *golden section*, whose supposedly aesthetic properties hark back to the ancient Pythagoreans. This optimal method of function minimization, the analog of the bisection method for finding zeros, is thus called the *golden section search*, summarized as follows:

Given, at each stage, a bracketing triplet of points, the next point to be tried is that which is a fraction 0.38197 into the larger of the two intervals (measuring from the central point of the triplet). If you start out with a bracketing triplet whose segments are not in the golden ratios, the procedure of choosing successive points at the golden mean point of the larger segment will quickly converge you to the proper, self-replicating ratios.

The golden section search guarantees that each new function evaluation will (after self-replicating ratios have been achieved) bracket the minimum to an interval

just 0.61803 times the size of the preceding interval. This is comparable to, but not quite as good as, the 0.50000 that holds when finding roots by bisection. Note that the convergence is *linear* (in the language of Chapter 9), meaning that successive significant figures are won linearly with additional function evaluations. In the next section we will give a superlinear method, where the rate at which successive significant figures are liberated increases with each successive function evaluation.

Routine for Initially Bracketing a Minimum

The preceding discussion has assumed that you are able to bracket the minimum in the first place. We consider this initial bracketing to be an essential part of any one-dimensional minimization. There are some one-dimensional algorithms that do not require a rigorous initial bracketing. However, we would *never* trade the secure feeling of *knowing* that a minimum is "in there somewhere" for the dubious reduction of function evaluations that these nonbracketing routines may promise. Please bracket your minima (or, for that matter, your zeros) before isolating them!

There is not much theory as to how to do this bracketing. Obviously you want to step downhill. But how far? We like to take larger and larger steps, starting with some (wild?) initial guess and then increasing the stepsize at each step either by a constant factor, or else by the result of a parabolic extrapolation of the preceding points that is designed to take us to the extrapolated turning point. It doesn't much matter if the steps get big. After all, we are stepping downhill, so we already have the left and middle points of the bracketing triplet. We just need to take a big enough step to stop the downhill trend and get a high third point.

Our standard routine is this:

```
#include <math.h>
#include "nrutil.h"
#define GOLD 1.618034
#define GLIMIT 100.0
#define TINY 1.0e-20
#define SHFT(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);
```

Here GOLD is the default ratio by which successive intervals are magnified; GLIMIT is the maximum magnification allowed for a parabolic-fit step.

```
void mnbrak(float *ax, float *bx, float *cx, float *fa, float *fb, float *fc,
    float (*func)(float))
```

Given a function func, and given distinct initial points ax and bx, this routine searches in the downhill direction (defined by the function as evaluated at the initial points) and returns new points ax, bx, cx that bracket a minimum of the function. Also returned are the function values at the three points, fa, fb, and fc.

```
{
    float ulim,u,r,q,fu,dum;

    *fa=(*func)(*ax);
    *fb=(*func)(*bx);
    if (*fb > *fa) {                          Switch roles of a and b so that we can go
        SHFT(dum,*ax,*bx,dum)                     downhill in the direction from a to b.
        SHFT(dum,*fb,*fa,dum)
    }
    *cx=(*bx)+GOLD*(*bx-*ax);                 First guess for c.
    *fc=(*func)(*cx);
    while (*fb > *fc) {                       Keep returning here until we bracket.
        r=(*bx-*ax)*(*fb-*fc);                Compute u by parabolic extrapolation from
        q=(*bx-*cx)*(*fb-*fa);                    a, b, c. TINY is used to prevent any pos-
        u=(*bx)-((*bx-*cx)*q-(*bx-*ax)*r)/        sible division by zero.
```

```
            (2.0*SIGN(FMAX(fabs(q-r),TINY),q-r));
        ulim=(*bx)+GLIMIT*(*cx-*bx);
        We won't go farther than this. Test various possibilities:
        if ((*bx-u)*(u-*cx) > 0.0) {          Parabolic u is between b and c: try it.
            fu=(*func)(u);
            if (fu < *fc) {                   Got a minimum between b and c.
                *ax=(*bx);
                *bx=u;
                *fa=(*fb);
                *fb=fu;
                return;
            } else if (fu > *fb) {            Got a minimum between between a and u.
                *cx=u;
                *fc=fu;
                return;
            }
            u=(*cx)+GOLD*(*cx-*bx);           Parabolic fit was no use. Use default mag-
            fu=(*func)(u);                        nification.
        } else if ((*cx-u)*(u-ulim) > 0.0) {          Parabolic fit is between c and its
            fu=(*func)(u);                                allowed limit.
            if (fu < *fc) {
                SHFT(*bx,*cx,u,*cx+GOLD*(*cx-*bx))
                SHFT(*fb,*fc,fu,(*func)(u))
            }
        } else if ((u-ulim)*(ulim-*cx) >= 0.0) {      Limit parabolic u to maximum
            u=ulim;                                       allowed value.
            fu=(*func)(u);
        } else {                              Reject parabolic u, use default magnifica-
            u=(*cx)+GOLD*(*cx-*bx);               tion.
            fu=(*func)(u);
        }
        SHFT(*ax,*bx,*cx,u)                   Eliminate oldest point and continue.
        SHFT(*fa,*fb,*fc,fu)
    }
}
```

(Because of the housekeeping involved in moving around three or four points and their function values, the above program ends up looking deceptively formidable. That is true of several other programs in this chapter as well. The underlying ideas, however, are quite simple.)

Routine for Golden Section Search

```
#include <math.h>
#define R 0.61803399                          The golden ratios.
#define C (1.0-R)
#define SHFT2(a,b,c) (a)=(b);(b)=(c);
#define SHFT3(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);

float golden(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 performs a golden section search for the minimum, isolating it to a fractional precision of about `tol`. The abscissa of the minimum is returned as `xmin`, and the minimum function value is returned as `golden`, the returned function value.

```
{
    float f1,f2,x0,x1,x2,x3;
```

```
    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

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

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

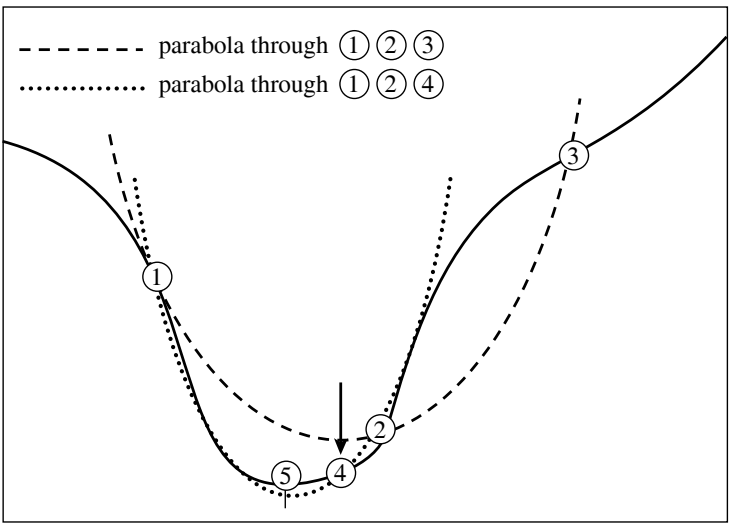


Figure 10.2.1. Convergence to a minimum by inverse parabolic interpolation. A parabola (dashed line) is drawn through the three original points 1,2,3 on the given function (solid line). The function is evaluated at the parabola's minimum, 4, which replaces point 3. A new parabola (dotted line) is drawn 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 $2 \times x \times \mathtt{tol}$ apart, with x (the best abscissa) at the midpoint of a and b, and therefore fractionally accurate to $\pm\mathtt{tol}$.

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");
    *xmin=x;                                            Never get here.
    return fx;
}
```

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.

```
#include <math.h>
#include "nrutil.h"
#define ITMAX 100
#define ZEPS 1.0e-10
#define MOV3(a,b,c, d,e,f) (a)=(d);(b)=(e);(c)=(f);

float dbrent(float ax, float bx, float cx, float (*f)(float),
    float (*df)(float), float tol, float *xmin)
```

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;                          Will be used as flags for whether pro-
    float a,b,d,d1,d2,du,dv,dw,dx,e=0.0;            posed steps are acceptable or not.
    float fu,fv,fw,fx,olde,tol1,tol2,u,u1,u2,v,w,x,xm;

    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 dou-
    for (iter=1;iter<=ITMAX;iter++) {              bled by the necessity of moving
        xm=0.5*(a+b);                              derivative values around as well
        tol1=tol*fabs(x)+ZEPS;                     as function values.
        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
            d2=d1;                                 value.
            if (dw != dx) d1=(w-x)*dx/(dx-dw);     Secant method with one point.
            if (dv != dx) d2=(v-x)*dx/(dx-dv);     And the other.
            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
                if (ok1 && ok2)                        both are acceptable, then take
                    d=(fabs(d1) < fabs(d2) ? d1 : d2);     the smallest one.
                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 {                           Bisect, not golden section.
                    d=0.5*(e=(dx >= 0.0 ? a-x : b-x));
                    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
                *xmin=x;                           direction takes us uphill, then
                return fx;                         we are done.
```

```
            }
        }
        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,

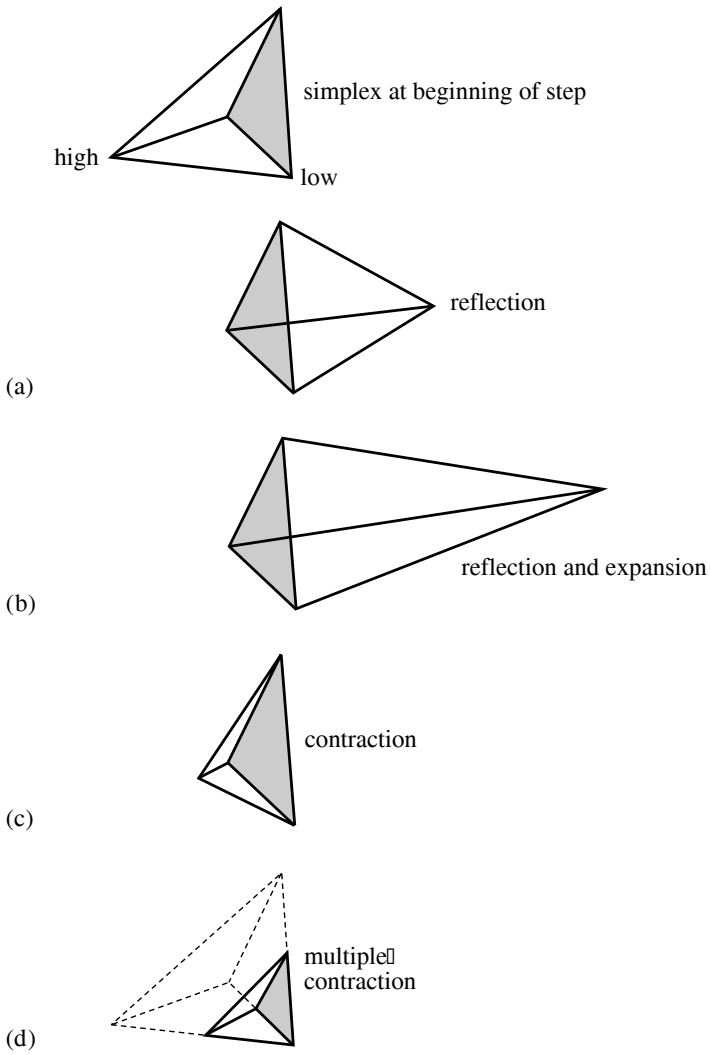


Figure 10.4.1. Possible outcomes for a step in the downhill simplex method. The simplex at the beginning of the step, here a tetrahedron, is shown, top. The simplex at the end of the step can be any one of (a) a reflection away from the high point, (b) a reflection and expansion away from the high point, (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 **P** in N-dimensional space, and proceed from there in some vector

direction **n**, then any function of N variables $f(\mathbf{P})$ can be minimized along the line **n** by our one-dimensional methods. One can dream up various multidimensional minimization methods that consist of sequences of such line minimizations. Different methods will differ only by how, at each stage, they choose the next direction **n** to try. All such methods presume the existence of a "black-box" sub-algorithm, which we might call `linmin` (given as an explicit routine at the end of this section), whose definition can be taken for now as

> `linmin`: Given as input the vectors **P** and **n**, and the function f, find the scalar λ that minimizes $f(\mathbf{P}+\lambda\mathbf{n})$. Replace **P** by $\mathbf{P}+\lambda\mathbf{n}$. Replace **n** by $\lambda\mathbf{n}$. Done.

All the minimization methods in this section and in the two sections following fall under this general schema of successive line minimizations. (The algorithm in §10.7 does not need very accurate line minimizations. Accordingly, it has its own approximate line minimization routine, `lnsrch`.) In this section we consider a class of methods whose choice of successive directions does not involve explicit computation of the function's gradient; the next two sections do require such gradient calculations. You will note that we need not specify whether `linmin` uses gradient information or not. That choice is up to you, and its optimization depends on your particular function. You would be crazy, however, to use gradients in `linmin` and *not* use them in the choice of directions, since in this latter role they can drastically reduce the total computational burden.

But what if, in your application, calculation of the gradient is out of the question. You might first think of this simple method: Take the unit vectors $\mathbf{e}_1, \mathbf{e}_2, \ldots \mathbf{e}_N$ as a *set of directions*. Using `linmin`, move along the first direction to its minimum, then *from there* along the second direction to *its* minimum, and so on, cycling through the whole set of directions as many times as necessary, until the function stops decreasing.

This simple method is actually not too bad for many functions. Even more interesting is why it *is* bad, i.e. very inefficient, for some other functions. Consider a function of two dimensions whose contour map (level lines) happens to define a long, narrow valley at some angle to the coordinate basis vectors (see Figure 10.5.1). Then the only way "down the length of the valley" going along the basis vectors at each stage is by a series of many tiny steps. More generally, in N dimensions, if the function's second derivatives are much larger in magnitude in some directions than in others, then many cycles through all N basis vectors will be required in order to get anywhere. This condition is not all that unusual; according to Murphy's Law, you should count on it.

Obviously what we need is a better set of directions than the $\mathbf{e}_i$'s. All *direction set methods* consist of prescriptions for updating the set of directions as the method proceeds, attempting to come up with a set which either (i) includes some very good directions that will take us far along narrow valleys, or else (more subtly) (ii) includes some number of "non-interfering" directions with the special property that minimization along one is not "spoiled" by subsequent minimization along another, so that interminable cycling through the set of directions can be avoided.

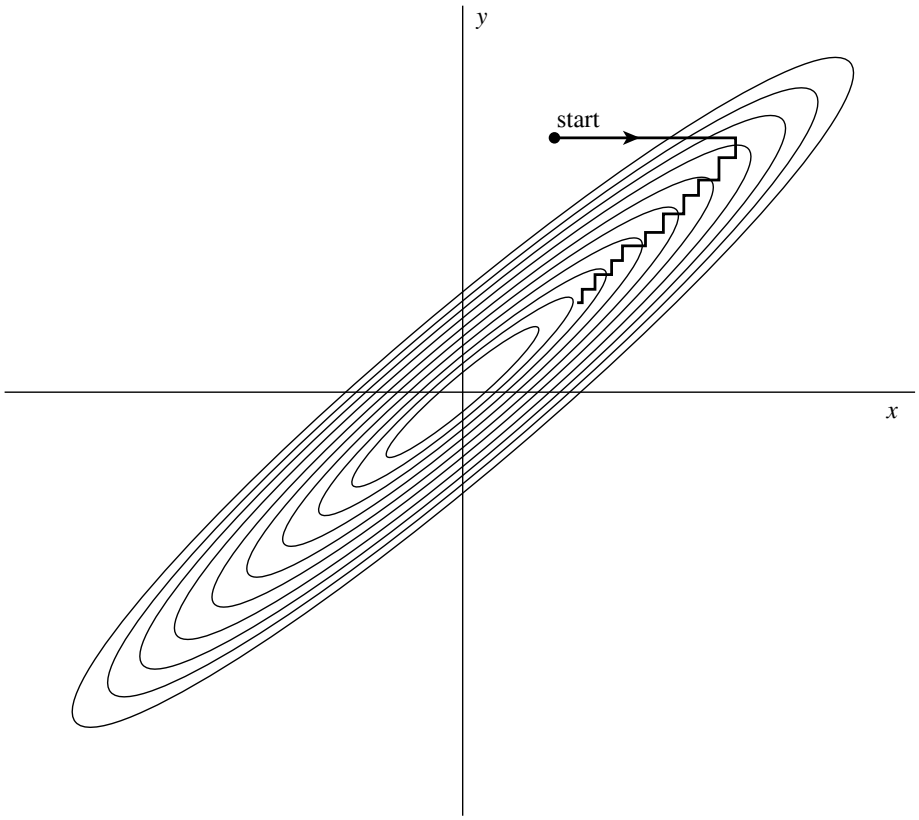


Figure 10.5.1. Successive minimizations along coordinate directions in a long, narrow "valley" (shown as contour lines). Unless the valley is optimally oriented, this method is extremely inefficient, taking many tiny steps to get to the minimum, crossing and re-crossing the principal axis.

Conjugate Directions

This concept of "non-interfering" directions, more conventionally called *conjugate directions*, is worth making mathematically explicit.

First, note that if we minimize a function along some direction $\mathbf{u}$, then the gradient of the function must be perpendicular to $\mathbf{u}$ at the line minimum; if not, then there would still be a nonzero directional derivative along $\mathbf{u}$.

Next take some particular point $\mathbf{P}$ as the origin of the coordinate system with coordinates $\mathbf{x}$. Then any function f can be approximated by its Taylor series

$$\begin{aligned} f(\mathbf{x}) &= f(\mathbf{P}) + \sum_i \frac{\partial f}{\partial x_i} x_i + \frac{1}{2} \sum_{i,j} \frac{\partial^2 f}{\partial x_i \partial x_j} x_i x_j + \cdots \\ &\approx c - \mathbf{b} \cdot \mathbf{x} + \frac{1}{2} \mathbf{x} \cdot \mathbf{A} \cdot \mathbf{x} \end{aligned} \tag{10.5.1}$$

where

$$c \equiv f(\mathbf{P}) \qquad \mathbf{b} \equiv -\nabla f|_{\mathbf{P}} \qquad [\mathbf{A}]_{ij} \equiv \left. \frac{\partial^2 f}{\partial x_i \partial x_j} \right|_{\mathbf{P}} \tag{10.5.2}$$

The matrix $\mathbf{A}$ whose components are the second partial derivative matrix of the function is called the *Hessian matrix* of the function at $\mathbf{P}$.

In the approximation of (10.5.1), the gradient of f is easily calculated as

$$\nabla f = \mathbf{A} \cdot \mathbf{x} - \mathbf{b} \tag{10.5.3}$$

(This implies that the gradient will vanish — the function will be at an extremum — at a value of $\mathbf{x}$ obtained by solving $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. This idea we will return to in §10.7!)

How does the gradient ∇f *change* as we move along some direction? Evidently

$$\delta(\nabla f) = \mathbf{A} \cdot (\delta \mathbf{x}) \tag{10.5.4}$$

Suppose that we have moved along some direction $\mathbf{u}$ to a minimum and now propose to move along some new direction $\mathbf{v}$. The condition that motion along $\mathbf{v}$ not *spoil* our minimization along $\mathbf{u}$ is just that the gradient stay perpendicular to $\mathbf{u}$, i.e., that the change in the gradient be perpendicular to $\mathbf{u}$. By equation (10.5.4) this is just

$$0 = \mathbf{u} \cdot \delta(\nabla f) = \mathbf{u} \cdot \mathbf{A} \cdot \mathbf{v} \tag{10.5.5}$$

When (10.5.5) holds for two vectors $\mathbf{u}$ and $\mathbf{v}$, they are said to be *conjugate*. When the relation holds pairwise for all members of a set of vectors, they are said to be a conjugate set. If you do successive line minimization of a function along a conjugate set of directions, then you don't need to redo any of those directions (unless, of course, you spoil things by minimizing along a direction that they are *not* conjugate to).

A triumph for a direction set method is to come up with a set of N linearly independent, mutually conjugate directions. Then, one pass of N line minimizations will put it exactly at the minimum of a quadratic form like (10.5.1). For functions f that are not exactly quadratic forms, it won't be exactly at the minimum; but repeated cycles of N line minimizations will in due course converge *quadratically* to the minimum.

Powell's Quadratically Convergent Method

Powell first discovered a direction set method that does produce N mutually conjugate directions. Here is how it goes: Initialize the set of directions $\mathbf{u}_i$ to the basis vectors,

$$\mathbf{u}_i = \mathbf{e}_i \qquad i = 1, \ldots, N \tag{10.5.6}$$

Now repeat the following sequence of steps ("basic procedure") until your function stops decreasing:

- Save your starting position as $\mathbf{P}_0$.
- For $i = 1, \ldots, N$, move $\mathbf{P}_{i-1}$ to the minimum along direction $\mathbf{u}_i$ and call this point $\mathbf{P}_i$.
- For $i = 1, \ldots, N-1$, set $\mathbf{u}_i \leftarrow \mathbf{u}_{i+1}$.
- Set $\mathbf{u}_N \leftarrow \mathbf{P}_N - \mathbf{P}_0$.
- Move $\mathbf{P}_N$ to the minimum along direction $\mathbf{u}_N$ and call this point $\mathbf{P}_0$.

Powell, in 1964, showed that, for a quadratic form like (10.5.1), k iterations of the above basic procedure produce a set of directions $\mathbf{u}_i$ whose last k members are mutually conjugate. Therefore, N iterations of the basic procedure, amounting to $N(N+1)$ line minimizations in all, will exactly minimize a quadratic form. Brent [1] gives proofs of these statements in accessible form.

Unfortunately, there is a problem with Powell's quadratically convergent algorithm. The procedure of throwing away, at each stage, $\mathbf{u}_1$ in favor of $\mathbf{P}_N - \mathbf{P}_0$ tends to produce sets of directions that "fold up on each other" and become linearly dependent. Once this happens, then the procedure finds the minimum of the function f only over a subspace of the full N-dimensional case; in other words, it gives the wrong answer. Therefore, the algorithm must not be used in the form given above.

There are a number of ways to fix up the problem of linear dependence in Powell's algorithm, among them:

1. You can reinitialize the set of directions $\mathbf{u}_i$ to the basis vectors $\mathbf{e}_i$ after every N or $N+1$ iterations of the basic procedure. This produces a serviceable method, which we commend to you if quadratic convergence is important for your application (i.e., if your functions are close to quadratic forms and if you desire high accuracy).

2. Brent points out that the set of directions can equally well be reset to the columns of any orthogonal matrix. Rather than throw away the information on conjugate directions already built up, he resets the direction set to calculated principal directions of the matrix $\mathbf{A}$ (which he gives a procedure for determining). The calculation is essentially a singular value decomposition algorithm (see §2.6). Brent has a number of other cute tricks up his sleeve, and his modification of Powell's method is probably the best presently known. Consult [1] for a detailed description and listing of the program. Unfortunately it is rather too elaborate for us to include here.

3. You can give up the property of quadratic convergence in favor of a more heuristic scheme (due to Powell) which tries to find a few good directions along narrow valleys instead of N necessarily conjugate directions. This is the method that we now implement. (It is also the version of Powell's method given in Acton [2], from which parts of the following discussion are drawn.)

Discarding the Direction of Largest Decrease

The fox and the grapes: Now that we are going to give up the property of quadratic convergence, was it so important after all? That depends on the function that you are minimizing. Some applications produce functions with long, twisty valleys. Quadratic convergence is of no particular advantage to a program which must slalom down the length of a valley floor that twists one way and another (and another, and another, ... – there are N dimensions!). Along the long direction, a quadratically convergent method is trying to extrapolate to the minimum of a parabola which just isn't (yet) there; while the conjugacy of the $N-1$ transverse directions keeps getting spoiled by the twists.

Sooner or later, however, we do arrive at an approximately ellipsoidal minimum (cf. equation 10.5.1 when $\mathbf{b}$, the gradient, is zero). Then, depending on how much accuracy we require, a method with quadratic convergence can save us several times N^2 extra line minimizations, since quadratic convergence *doubles* the number of significant figures at each iteration.

The basic idea of our now-modified Powell's method is still to take $\mathbf{P}_N - \mathbf{P}_0$ as a new direction; it is, after all, the average direction moved after trying all N possible directions. For a valley whose long direction is twisting slowly, this direction is likely to give us a good run along the new long direction. The change is to discard the old direction along which the function f made its *largest decrease*. This seems paradoxical, since that direction was the *best* of the previous iteration. However, it is also likely to be a major component of the new direction that we are adding, so dropping it gives us the best chance of avoiding a buildup of linear dependence.

There are a couple of exceptions to this basic idea. Sometimes it is better *not* to add a new direction at all. Define

$$f_0 \equiv f(\mathbf{P}_0) \qquad f_N \equiv f(\mathbf{P}_N) \qquad f_E \equiv f(2\mathbf{P}_N - \mathbf{P}_0) \tag{10.5.7}$$

Here f_E is the function value at an "extrapolated" point somewhat further along the proposed new direction. Also define Δf to be the magnitude of the largest decrease along one particular direction of the present basic procedure iteration. (Δf is a positive number.) Then:

1. If $f_E \geq f_0$, then keep the old set of directions for the next basic procedure, because the average direction $\mathbf{P}_N - \mathbf{P}_0$ is all played out.

2. If $2\,(f_0 - 2f_N + f_E)\,[(f_0 - f_N) - \Delta f]^2 \geq (f_0 - f_E)^2 \Delta f$, then keep the old set of directions for the next basic procedure, because either (i) the decrease along the average direction was not primarily due to any single direction's decrease, or (ii) there is a substantial second derivative along the average direction and we seem to be near to the bottom of its minimum.

The following routine implements Powell's method in the version just described. In the routine, `xi` is the matrix whose columns are the set of directions $\mathbf{n}_i$; otherwise the correspondence of notation should be self-evident.

```
#include <math.h>
#include "nrutil.h"
#define TINY 1.0e-25                    A small number.
#define ITMAX 200                       Maximum allowed iterations.

void powell(float p[], float **xi, int n, float ftol, int *iter, float *fret,
    float (*func)(float []))
```

Minimization of a function `func` of `n` variables. Input consists of an initial starting point `p[1..n]`; an initial matrix `xi[1..n][1..n]`, whose columns contain the initial set of directions (usually the `n` unit vectors); and `ftol`, the fractional tolerance in the function value such that failure to decrease by more than this amount on one iteration signals doneness. On output, `p` is set to the best point found, `xi` is the then-current direction set, `fret` is the returned function value at `p`, and `iter` is the number of iterations taken. The routine `linmin` is used.

```
{
    void linmin(float p[], float xi[], int n, float *fret,
        float (*func)(float []));
    int i,ibig,j;
    float del,fp,fptt,t,*pt,*ptt,*xit;

    pt=vector(1,n);
    ptt=vector(1,n);
    xit=vector(1,n);
    *fret=(*func)(p);
    for (j=1;j<=n;j++) pt[j]=p[j];              Save the initial point.
    for (*iter=1;;++(*iter)) {
        fp=(*fret);
        ibig=0;
```

```
        del=0.0;                        Will be the biggest function decrease.
        for (i=1;i<=n;i++) {            In each iteration, loop over all directions in the set.
            for (j=1;j<=n;j++) xit[j]=xi[j][i];       Copy the direction,
            fptt=(*fret);
            linmin(p,xit,n,fret,func);          minimize along it,
            if (fptt-(*fret) > del) {           and record it if it is the largest decrease
                del=fptt-(*fret);                   so far.
                ibig=i;
            }
        }
        if (2.0*(fp-(*fret)) <= ftol*(fabs(fp)+fabs(*fret))+TINY) {
            free_vector(xit,1,n);               Termination criterion.
            free_vector(ptt,1,n);
            free_vector(pt,1,n);
            return;
        }
        if (*iter == ITMAX) nrerror("powell exceeding maximum iterations.");
        for (j=1;j<=n;j++) {                    Construct the extrapolated point and the
            ptt[j]=2.0*p[j]-pt[j];                  average direction moved. Save the
            xit[j]=p[j]-pt[j];                      old starting point.
            pt[j]=p[j];
        }
        fptt=(*func)(ptt);                      Function value at extrapolated point.
        if (fptt < fp) {
            t=2.0*(fp-2.0*(*fret)+fptt)*SQR(fp-(*fret)-del)-del*SQR(fp-fptt);
            if (t < 0.0) {
                linmin(p,xit,n,fret,func);      Move to the minimum of the new direc-
                for (j=1;j<=n;j++) {                tion, and save the new direction.
                    xi[j][ibig]=xi[j][n];
                    xi[j][n]=xit[j];
                }
            }
        }
    }                                           Back for another iteration.
}
```

Implementation of Line Minimization

Make no mistake, there is a *right* way to implement `linmin`: It is to use the *methods* of one-dimensional minimization described in §10.1–§10.3, but to rewrite the programs of those sections so that their bookkeeping is done on vector-valued points **P** (all lying along a given direction **n**) rather than scalar-valued abscissas x. That straightforward task produces long routines densely populated with "`for(k=1;k<=n;k++)`" loops.

We do not have space to include such routines in this book. Our `linmin`, which works just fine, is instead a kind of bookkeeping swindle. It constructs an "artificial" function of one variable called `f1dim`, which is the value of your function, say, `func`, along the line going through the point `p` in the direction `xi`. `linmin` calls our familiar one-dimensional routines `mnbrak` (§10.1) and `brent` (§10.3) and instructs them to minimize `f1dim`. `linmin` communicates with `f1dim` "over the head" of `mnbrak` and `brent`, through global (external) variables. That is also how it passes to `f1dim` a pointer to your user-supplied function.

The only thing inefficient about `linmin` is this: Its use as an interface between a multidimensional minimization strategy and a one-dimensional minimization routine results in some unnecessary copying of vectors hither and yon. That should not

normally be a significant addition to the overall computational burden, but we cannot disguise its inelegance.

```
#include "nrutil.h"
#define TOL 2.0e-4                      Tolerance passed to brent.

int ncom;                               Global variables communicate with f1dim.
float *pcom,*xicom,(*nrfunc)(float []);

void linmin(float p[], float xi[], int n, float *fret, float (*func)(float []))
Given an n-dimensional point p[1..n] and an n-dimensional direction xi[1..n], moves and
resets p to where the function func(p) takes on a minimum along the direction xi from p,
and replaces xi by the actual vector displacement that p was moved. Also returns as fret
the value of func at the returned location p. This is actually all accomplished by calling the
routines mnbrak and brent.
{
    float brent(float ax, float bx, float cx,
        float (*f)(float), float tol, float *xmin);
    float f1dim(float x);
    void mnbrak(float *ax, float *bx, float *cx, float *fa, float *fb,
        float *fc, float (*func)(float));
    int j;
    float xx,xmin,fx,fb,fa,bx,ax;

    ncom=n;                             Define the global variables.
    pcom=vector(1,n);
    xicom=vector(1,n);
    nrfunc=func;
    for (j=1;j<=n;j++) {
        pcom[j]=p[j];
        xicom[j]=xi[j];
    }
    ax=0.0;                             Initial guess for brackets.
    xx=1.0;
    mnbrak(&ax,&xx,&bx,&fa,&fx,&fb,f1dim);
    *fret=brent(ax,xx,bx,f1dim,TOL,&xmin);
    for (j=1;j<=n;j++) {                Construct the vector results to return.
        xi[j] *= xmin;
        p[j] += xi[j];
    }
    free_vector(xicom,1,n);
    free_vector(pcom,1,n);
}



#include "nrutil.h"

extern int ncom;                            Defined in linmin.
extern float *pcom,*xicom,(*nrfunc)(float []);

float f1dim(float x)
Must accompany linmin.
{
    int j;
    float f,*xt;

    xt=vector(1,ncom);
    for (j=1;j<=ncom;j++) xt[j]=pcom[j]+x*xicom[j];
    f=(*nrfunc)(xt);
    free_vector(xt,1,ncom);
    return f;
}
```

CITED REFERENCES AND FURTHER READING:

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 7. [1]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 464–467. [2]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), pp. 259–262.

10.6 Conjugate Gradient Methods in Multidimensions

We consider now the case where you are able to calculate, at a given N-dimensional point $\mathbf{P}$, not just the value of a function $f(\mathbf{P})$ but also the gradient (vector of first partial derivatives) $\nabla f(\mathbf{P})$.

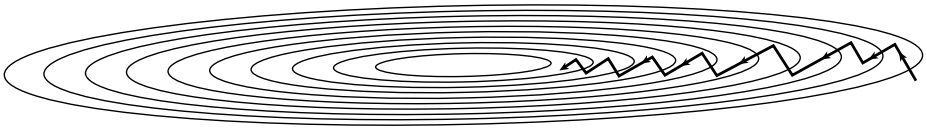
A rough counting argument will show how advantageous it is to use the gradient information: Suppose that the function f is roughly approximated as a quadratic form, as above in equation (10.5.1),

$$f(\mathbf{x}) \approx c - \mathbf{b} \cdot \mathbf{x} + \frac{1}{2} \mathbf{x} \cdot \mathbf{A} \cdot \mathbf{x} \tag{10.6.1}$$

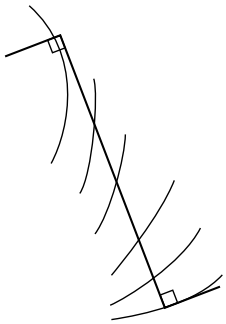
Then the number of unknown parameters in f is equal to the number of free parameters in $\mathbf{A}$ and $\mathbf{b}$, which is $\frac{1}{2}N(N+1)$, which we see to be of order N^2. Changing any one of these parameters can move the location of the minimum. Therefore, we should not expect to be able to *find* the minimum until we have collected an equivalent information content, of order N^2 numbers.

In the direction set methods of §10.5, we collected the necessary information by making on the order of N^2 separate line minimizations, each requiring "a few" (but sometimes a *big* few!) function evaluations. Now, each evaluation of the gradient will bring us N new components of information. If we use them wisely, we should need to make only of order N separate line minimizations. That is in fact the case for the algorithms in this section and the next.

A factor of N improvement in computational speed is not necessarily implied. As a rough estimate, we might imagine that the calculation of *each component* of the gradient takes about as long as evaluating the function itself. In that case there will be of order N^2 equivalent function evaluations both with and without gradient information. Even if the advantage is not of order N, however, it is nevertheless quite substantial: (i) Each calculated component of the gradient will typically save not just one function evaluation, but a number of them, equivalent to, say, a whole line minimization. (ii) There is often a high degree of redundancy in the formulas for the various components of a function's gradient; when this is so, especially when there is also redundancy with the calculation of the function, then the calculation of the gradient may cost significantly less than N function evaluations.



(a)



(b)

Figure 10.6.1. (a) Steepest descent method in a long, narrow "valley." While more efficient than the strategy of Figure 10.5.1, steepest descent is nonetheless an inefficient strategy, taking many steps to reach the valley floor. (b) Magnified view of one step: A step starts off in the local gradient direction, perpendicular to the contour lines, and traverses a straight line until a local minimum is reached, where the traverse is parallel to the local contour lines.

A common beginner's error is to assume that any reasonable way of incorporating gradient information should be about as good as any other. This line of thought leads to the following *not very good* algorithm, the *steepest descent method*:

> Steepest Descent: Start at a point $\mathbf{P}_0$. As many times as needed, move from point $\mathbf{P}_i$ to the point $\mathbf{P}_{i+1}$ by minimizing along the line from $\mathbf{P}_i$ in the direction of the local downhill gradient $-\nabla f(\mathbf{P}_i)$.

The problem with the steepest descent method (which, incidentally, goes back to Cauchy), is similar to the problem that was shown in Figure 10.5.1. The method will perform many small steps in going down a long, narrow valley, even if the valley is a perfect quadratic form. You might have hoped that, say in two dimensions, your first step would take you to the valley floor, the second step directly down the long axis; but remember that the new gradient at the minimum point of any line minimization is perpendicular to the direction just traversed. Therefore, with the steepest descent method, you *must* make a right angle turn, which does *not*, in general, take you to the minimum. (See Figure 10.6.1.)

Just as in the discussion that led up to equation (10.5.5), we really want a way of proceeding not down the new gradient, but rather in a direction that is somehow constructed to be *conjugate* to the old gradient, and, insofar as possible, to all previous directions traversed. Methods that accomplish this construction are called *conjugate gradient* methods.

In §2.7 we discussed the conjugate gradient method as a technique for solving linear algebraic equations by minimizing a quadratic form. That formalism can also be applied to the problem of minimizing a function *approximated* by the quadratic

form (10.6.1). Recall that, starting with an arbitrary initial vector $\mathbf{g}_0$ and letting $\mathbf{h}_0 = \mathbf{g}_0$, the conjugate gradient method constructs two sequences of vectors from the recurrence

$$\mathbf{g}_{i+1} = \mathbf{g}_i - \lambda_i \mathbf{A} \cdot \mathbf{h}_i \qquad \mathbf{h}_{i+1} = \mathbf{g}_{i+1} + \gamma_i \mathbf{h}_i \qquad i = 0, 1, 2, \ldots \tag{10.6.2}$$

The vectors satisfy the orthogonality and conjugacy conditions

$$\mathbf{g}_i \cdot \mathbf{g}_j = 0 \qquad \mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_j = 0 \qquad \mathbf{g}_i \cdot \mathbf{h}_j = 0 \qquad j < i \tag{10.6.3}$$

The scalars λ_i and γ_i are given by

$$\lambda_i = \frac{\mathbf{g}_i \cdot \mathbf{g}_i}{\mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_i} = \frac{\mathbf{g}_i \cdot \mathbf{h}_i}{\mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_i} \tag{10.6.4}$$

$$\gamma_i = \frac{\mathbf{g}_{i+1} \cdot \mathbf{g}_{i+1}}{\mathbf{g}_i \cdot \mathbf{g}_i} \tag{10.6.5}$$

Equations (10.6.2)–(10.6.5) are simply equations (2.7.32)–(2.7.35) for a symmetric $\mathbf{A}$ in a new notation. (A self-contained derivation of these results in the context of function minimization is given by Polak [1].)

Now suppose that we knew the Hessian matrix $\mathbf{A}$ in equation (10.6.1). Then we could use the construction (10.6.2) to find successively conjugate directions $\mathbf{h}_i$ along which to line-minimize. After N such, we would efficiently have arrived at the minimum of the quadratic form. But we don't know $\mathbf{A}$.

Here is a remarkable theorem to save the day: Suppose we happen to have $\mathbf{g}_i = -\nabla f(\mathbf{P}_i)$, for some point $\mathbf{P}_i$, where f is of the form (10.6.1). Suppose that we proceed from $\mathbf{P}_i$ along the direction $\mathbf{h}_i$ to the local minimum of f located at some point $\mathbf{P}_{i+1}$ and then set $\mathbf{g}_{i+1} = -\nabla f(\mathbf{P}_{i+1})$. Then, this $\mathbf{g}_{i+1}$ is the same vector as would have been constructed by equation (10.6.2). (And we have constructed it without knowledge of $\mathbf{A}$!)

Proof: By equation (10.5.3), $\mathbf{g}_i = -\mathbf{A} \cdot \mathbf{P}_i + \mathbf{b}$, and

$$\mathbf{g}_{i+1} = -\mathbf{A} \cdot (\mathbf{P}_i + \lambda \mathbf{h}_i) + \mathbf{b} = \mathbf{g}_i - \lambda \mathbf{A} \cdot \mathbf{h}_i \tag{10.6.6}$$

with λ chosen to take us to the line minimum. But at the line minimum $\mathbf{h}_i \cdot \nabla f = -\mathbf{h}_i \cdot \mathbf{g}_{i+1} = 0$. This latter condition is easily combined with (10.6.6) to solve for λ. The result is exactly the expression (10.6.4). But with this value of λ, (10.6.6) is the same as (10.6.2), q.e.d.

We have, then, the basis of an algorithm that requires neither knowledge of the Hessian matrix $\mathbf{A}$, nor even the storage necessary to store such a matrix. A sequence of directions $\mathbf{h}_i$ is constructed, using only line minimizations, evaluations of the gradient vector, and an auxiliary vector to store the latest in the sequence of $\mathbf{g}$'s.

The algorithm described so far is the original Fletcher-Reeves version of the conjugate gradient algorithm. Later, Polak and Ribiere introduced one tiny, but sometimes significant, change. They proposed using the form

$$\gamma_i = \frac{(\mathbf{g}_{i+1} - \mathbf{g}_i) \cdot \mathbf{g}_{i+1}}{\mathbf{g}_i \cdot \mathbf{g}_i} \tag{10.6.7}$$

instead of equation (10.6.5). "Wait," you say, "aren't they equal by the orthogonality conditions (10.6.3)?" They are equal for exact quadratic forms. In the real world, however, your function is not exactly a quadratic form. Arriving at the supposed minimum of the quadratic form, you may still need to proceed for another set of iterations. There is some evidence [2] that the Polak-Ribiere formula accomplishes the transition to further iterations more gracefully: When it runs out of steam, it tends to reset **h** to be down the local gradient, which is equivalent to beginning the conjugate-gradient procedure anew.

The following routine implements the Polak-Ribiere variant, which we recommend; but changing one program line, as shown, will give you Fletcher-Reeves. The routine presumes the existence of a function `func(p)`, where `p[1..n]` is a vector of length `n`, and also presumes the existence of a function `dfunc(p,df)` that sets the vector gradient `df[1..n]` evaluated at the input point `p`.

The routine calls `linmin` to do the line minimizations. As already discussed, you may wish to use a modified version of `linmin` that uses `dbrent` instead of `brent`, i.e., that uses the gradient in doing the line minimizations. See note below.

```
#include <math.h>
#include "nrutil.h"
#define ITMAX 200
#define EPS 1.0e-10
```
Here ITMAX is the maximum allowed number of iterations, while EPS is a small number to rectify the special case of converging to exactly zero function value.
```
#define FREEALL free_vector(xi,1,n);free_vector(h,1,n);free_vector(g,1,n);

void frprmn(float p[], int n, float ftol, int *iter, float *fret,
    float (*func)(float []), void (*dfunc)(float [], float []))
```
Given a starting point `p[1..n]`, Fletcher-Reeves-Polak-Ribiere minimization is performed on a function `func`, using its gradient as calculated by a routine `dfunc`. The convergence tolerance on the function value is input as `ftol`. Returned quantities are `p` (the location of the minimum), `iter` (the number of iterations that were performed), and `fret` (the minimum value of the function). The routine `linmin` is called to perform line minimizations.
```
{
    void linmin(float p[], float xi[], int n, float *fret,
        float (*func)(float []));
    int j,its;
    float gg,gam,fp,dgg;
    float *g,*h,*xi;

    g=vector(1,n);
    h=vector(1,n);
    xi=vector(1,n);
    fp=(*func)(p);                          Initializations.
    (*dfunc)(p,xi);
    for (j=1;j<=n;j++) {
        g[j] = -xi[j];
        xi[j]=h[j]=g[j];
    }
    for (its=1;its<=ITMAX;its++) {          Loop over iterations.
        *iter=its;
        linmin(p,xi,n,fret,func);           Next statement is the normal return:
        if (2.0*fabs(*fret-fp) <= ftol*(fabs(*fret)+fabs(fp)+EPS)) {
            FREEALL
            return;
        }
        fp= *fret;
        (*dfunc)(p,xi);
        dgg=gg=0.0;
        for (j=1;j<=n;j++) {
```

```
            gg += g[j]*g[j];
/*          dgg += xi[j]*xi[j];     */          This statement for Fletcher-Reeves.
            dgg += (xi[j]+g[j])*xi[j];          This statement for Polak-Ribiere.
        }
        if (gg == 0.0) {                        Unlikely. If gradient is exactly zero then
            FREEALL                                 we are already done.
            return;
        }
        gam=dgg/gg;
        for (j=1;j<=n;j++) {
            g[j] = -xi[j];
            xi[j]=h[j]=g[j]+gam*h[j];
        }
    }
    nrerror("Too many iterations in frprmn");
}
```

Note on Line Minimization Using Derivatives

Kindly reread the last part of §10.5. We here want to do the same thing, but using derivative information in performing the line minimization.

The modified version of `linmin`, called `dlinmin`, and its required companion routine `df1dim` follow:

```
#include "nrutil.h"
#define TOL 2.0e-4                  Tolerance passed to dbrent.

int ncom;                           Global variables communicate with df1dim.
float *pcom,*xicom,(*nrfunc)(float []);
void (*nrdfun)(float [], float []);

void dlinmin(float p[], float xi[], int n, float *fret, float (*func)(float []),
    void (*dfunc)(float [], float []))
Given an n-dimensional point p[1..n] and an n-dimensional direction xi[1..n], moves and
resets p to where the function func(p) takes on a minimum along the direction xi from p,
and replaces xi by the actual vector displacement that p was moved. Also returns as fret
the value of func at the returned location p. This is actually all accomplished by calling the
routines mnbrak and dbrent.
{
    float dbrent(float ax, float bx, float cx,
        float (*f)(float), float (*df)(float), float tol, float *xmin);
    float f1dim(float x);
    float df1dim(float x);
    void mnbrak(float *ax, float *bx, float *cx, float *fa, float *fb,
        float *fc, float (*func)(float));
    int j;
    float xx,xmin,fx,fb,fa,bx,ax;

    ncom=n;                         Define the global variables.
    pcom=vector(1,n);
    xicom=vector(1,n);
    nrfunc=func;
    nrdfun=dfunc;
    for (j=1;j<=n;j++) {
        pcom[j]=p[j];
        xicom[j]=xi[j];
    }
    ax=0.0;                         Initial guess for brackets.
    xx=1.0;
    mnbrak(&ax,&xx,&bx,&fa,&fx,&fb,f1dim);
```

```
    *fret=dbrent(ax,xx,bx,f1dim,df1dim,TOL,&xmin);
    for (j=1;j<=n;j++) {          Construct the vector results to return.
        xi[j] *= xmin;
        p[j] += xi[j];
    }
    free_vector(xicom,1,n);
    free_vector(pcom,1,n);
}
```

```
#include "nrutil.h"

extern int ncom;                          Defined in dlinmin.
extern float *pcom,*xicom,(*nrfunc)(float []);
extern void (*nrdfun)(float [], float []);

float df1dim(float x)
{
    int j;
    float df1=0.0;
    float *xt,*df;

    xt=vector(1,ncom);
    df=vector(1,ncom);
    for (j=1;j<=ncom;j++) xt[j]=pcom[j]+x*xicom[j];
    (*nrdfun)(xt,df);
    for (j=1;j<=ncom;j++) df1 += df[j]*xicom[j];
    free_vector(df,1,ncom);
    free_vector(xt,1,ncom);
    return df1;
}
```

CITED REFERENCES AND FURTHER READING:

Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press), §2.3. [1]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.1.7 (by K.W. Brodlie). [2]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §8.7.

10.7 Variable Metric Methods in Multidimensions

The goal of *variable metric* methods, which are sometimes called *quasi-Newton* methods, is not different from the goal of conjugate gradient methods: to accumulate information from successive line minimizations so that N such line minimizations lead to the exact minimum of a quadratic form in N dimensions. In that case, the method will also be quadratically convergent for more general smooth functions.

Both variable metric and conjugate gradient methods require that you are able to compute your function's gradient, or first partial derivatives, at arbitrary points. The variable metric approach differs from the conjugate gradient in the way that it stores

and updates the information that is accumulated. Instead of requiring intermediate storage on the order of N, the number of dimensions, it requires a matrix of size $N \times N$. Generally, for any moderate N, this is an entirely trivial disadvantage.

On the other hand, there is not, as far as we know, any overwhelming advantage that the variable metric methods hold over the conjugate gradient techniques, except perhaps a historical one. Developed somewhat earlier, and more widely propagated, the variable metric methods have by now developed a wider constituency of satisfied users. Likewise, some fancier implementations of variable metric methods (going beyond the scope of this book, see below) have been developed to a greater level of sophistication on issues like the minimization of roundoff error, handling of special conditions, and so on. *We* tend to use variable metric rather than conjugate gradient, but we have no reason to urge this habit on you.

Variable metric methods come in two main flavors. One is the *Davidon-Fletcher-Powell (DFP)* algorithm (sometimes referred to as simply *Fletcher-Powell*). The other goes by the name *Broyden-Fletcher-Goldfarb-Shanno (BFGS)*. The BFGS and DFP schemes differ only in details of their roundoff error, convergence tolerances, and similar "dirty" issues which are outside of our scope [1,2]. However, it has become generally recognized that, empirically, the BFGS scheme is superior in these details. We will implement BFGS in this section.

As before, we imagine that our arbitrary function $f(\mathbf{x})$ can be locally approximated by the quadratic form of equation (10.6.1). We don't, however, have any information about the values of the quadratic form's parameters $\mathbf{A}$ and $\mathbf{b}$, except insofar as we can glean such information from our function evaluations and line minimizations.

The basic idea of the variable metric method is to build up, iteratively, a good approximation to the inverse Hessian matrix $\mathbf{A}^{-1}$, that is, to construct a sequence of matrices $\mathbf{H}_i$ with the property,

$$\lim_{i \to \infty} \mathbf{H}_i = \mathbf{A}^{-1} \tag{10.7.1}$$

Even better if the limit is achieved after N iterations instead of ∞.

The reason that variable metric methods are sometimes called quasi-Newton methods can now be explained. Consider finding a minimum by using Newton's method to search for a zero of the gradient of the function. Near the current point $\mathbf{x}_i$, we have to second order

$$f(\mathbf{x}) = f(\mathbf{x}_i) + (\mathbf{x} - \mathbf{x}_i) \cdot \nabla f(\mathbf{x}_i) + \tfrac{1}{2}(\mathbf{x} - \mathbf{x}_i) \cdot \mathbf{A} \cdot (\mathbf{x} - \mathbf{x}_i) \tag{10.7.2}$$

so

$$\nabla f(\mathbf{x}) = \nabla f(\mathbf{x}_i) + \mathbf{A} \cdot (\mathbf{x} - \mathbf{x}_i) \tag{10.7.3}$$

In Newton's method we set $\nabla f(\mathbf{x}) = 0$ to determine the next iteration point:

$$\mathbf{x} - \mathbf{x}_i = -\mathbf{A}^{-1} \cdot \nabla f(\mathbf{x}_i) \tag{10.7.4}$$

The left-hand side is the finite step we need take to get to the exact minimum; the right-hand side is known once we have accumulated an accurate $\mathbf{H} \approx \mathbf{A}^{-1}$.

The "quasi" in quasi-Newton is because we don't use the actual Hessian matrix of f, but instead use our current approximation of it. This is often *better* than

using the true Hessian. We can understand this paradoxical result by considering the *descent directions* of f at $\mathbf{x}_i$. These are the directions $\mathbf{p}$ along which f decreases: $\nabla f \cdot \mathbf{p} < 0$. For the Newton direction (10.7.4) to be a descent direction, we must have

$$\nabla f(\mathbf{x}_i) \cdot (\mathbf{x} - \mathbf{x}_i) = -(\mathbf{x} - \mathbf{x}_i) \cdot \mathbf{A} \cdot (\mathbf{x} - \mathbf{x}_i) < 0 \tag{10.7.5}$$

which is true if $\mathbf{A}$ is positive definite. In general, far from a minimum, we have no guarantee that the Hessian is positive definite. Taking the actual Newton step with the real Hessian can move us to points where the function is *increasing* in value. The idea behind quasi-Newton methods is to start with a positive definite, symmetric approximation to $\mathbf{A}$ (usually the unit matrix) and build up the approximating $\mathbf{H}_i$'s in such a way that the matrix $\mathbf{H}_i$ remains positive definite and symmetric. Far from the minimum, this guarantees that we always move in a downhill direction. Close to the minimum, the updating formula approaches the true Hessian and we enjoy the quadratic convergence of Newton's method.

When we are not close enough to the minimum, taking the full Newton step $\mathbf{p}$ even with a positive definite $\mathbf{A}$ need not decrease the function; we may move too far for the quadratic approximation to be valid. All we are guaranteed is that *initially* f decreases as we move in the Newton direction. Once again we can use the backtracking strategy described in §9.7 to choose a step along the *direction* of the Newton step $\mathbf{p}$, but not necessarily all the way.

We won't rigorously derive the DFP algorithm for taking $\mathbf{H}_i$ into $\mathbf{H}_{i+1}$; you can consult [3] for clear derivations. Following Brodlie (in [2]), we will give the following heuristic motivation of the procedure.

Subtracting equation (10.7.4) at $\mathbf{x}_{i+1}$ from that same equation at $\mathbf{x}_i$ gives

$$\mathbf{x}_{i+1} - \mathbf{x}_i = \mathbf{A}^{-1} \cdot (\nabla f_{i+1} - \nabla f_i) \tag{10.7.6}$$

where $\nabla f_j \equiv \nabla f(\mathbf{x}_j)$. Having made the step from $\mathbf{x}_i$ to $\mathbf{x}_{i+1}$, we might reasonably want to require that the new approximation $\mathbf{H}_{i+1}$ satisfy (10.7.6) as if it were actually $\mathbf{A}^{-1}$, that is,

$$\mathbf{x}_{i+1} - \mathbf{x}_i = \mathbf{H}_{i+1} \cdot (\nabla f_{i+1} - \nabla f_i) \tag{10.7.7}$$

We might also imagine that the updating formula should be of the form $\mathbf{H}_{i+1} = \mathbf{H}_i + \text{correction}$.

What "objects" are around out of which to construct a correction term? Most notable are the two vectors $\mathbf{x}_{i+1} - \mathbf{x}_i$ and $\nabla f_{i+1} - \nabla f_i$; and there is also $\mathbf{H}_i$. There are not infinitely many natural ways of making a matrix out of these objects, especially if (10.7.7) must hold! One such way, the *DFP updating formula*, is

$$\begin{aligned} \mathbf{H}_{i+1} = \mathbf{H}_i &+ \frac{(\mathbf{x}_{i+1} - \mathbf{x}_i) \otimes (\mathbf{x}_{i+1} - \mathbf{x}_i)}{(\mathbf{x}_{i+1} - \mathbf{x}_i) \cdot (\nabla f_{i+1} - \nabla f_i)} \\ &- \frac{[\mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)] \otimes [\mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)]}{(\nabla f_{i+1} - \nabla f_i) \cdot \mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)} \end{aligned} \tag{10.7.8}$$

where $\otimes$ denotes the "outer" or "direct" product of two vectors, a matrix: The ij component of $\mathbf{u} \otimes \mathbf{v}$ is $u_i v_j$. (You might want to verify that 10.7.8 does satisfy 10.7.7.)

The *BFGS updating formula* is exactly the same, but with one additional term,

$$\cdots + [(\nabla f_{i+1} - \nabla f_i) \cdot \mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)]\, \mathbf{u} \otimes \mathbf{u} \tag{10.7.9}$$

where $\mathbf{u}$ is defined as the vector

$$\mathbf{u} \equiv \frac{(\mathbf{x}_{i+1} - \mathbf{x}_i)}{(\mathbf{x}_{i+1} - \mathbf{x}_i) \cdot (\nabla f_{i+1} - \nabla f_i)} - \frac{\mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)}{(\nabla f_{i+1} - \nabla f_i) \cdot \mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)} \tag{10.7.10}$$

(You might also verify that this satisfies 10.7.7.)

You will have to take on faith — or else consult [3] for details of — the "deep" result that equation (10.7.8), with or without (10.7.9), does in fact converge to $\mathbf{A}^{-1}$ in N steps, if f is a quadratic form.

Here now is the routine `dfpmin` that implements the quasi-Newton method, and uses `lnsrch` from §9.7. As mentioned at the end of `newt` in §9.7, this algorithm can fail if your variables are badly scaled.

```
#include <math.h>
#include "nrutil.h"
#define ITMAX 200                                   Maximum allowed number of iterations.
#define EPS 3.0e-8                                  Machine precision.
#define TOLX (4*EPS)                                Convergence criterion on x values.
#define STPMX 100.0                                 Scaled maximum step length allowed in
                                                        line searches.
#define FREEALL free_vector(xi,1,n);free_vector(pnew,1,n); \
free_matrix(hessin,1,n,1,n);free_vector(hdg,1,n);free_vector(g,1,n); \
free_vector(dg,1,n);

void dfpmin(float p[], int n, float gtol, int *iter, float *fret,
    float(*func)(float []), void (*dfunc)(float [], float []))
Given a starting point p[1..n] that is a vector of length n, the Broyden-Fletcher-Goldfarb-
Shanno variant of Davidon-Fletcher-Powell minimization is performed on a function func, using
its gradient as calculated by a routine dfunc. The convergence requirement on zeroing the
gradient is input as gtol. Returned quantities are p[1..n] (the location of the minimum),
iter (the number of iterations that were performed), and fret (the minimum value of the
function). The routine lnsrch is called to perform approximate line minimizations.
{
    void lnsrch(int n, float xold[], float fold, float g[], float p[], float x[],
        float *f, float stpmax, int *check, float (*func)(float []));
    int check,i,its,j;
    float den,fac,fad,fae,fp,stpmax,sum=0.0,sumdg,sumxi,temp,test;
    float *dg,*g,*hdg,**hessin,*pnew,*xi;

    dg=vector(1,n);
    g=vector(1,n);
    hdg=vector(1,n);
    hessin=matrix(1,n,1,n);
    pnew=vector(1,n);
    xi=vector(1,n);
    fp=(*func)(p);                                  Calculate starting function value and gra-
    (*dfunc)(p,g);                                      dient,
    for (i=1;i<=n;i++) {                            and initialize the inverse Hessian to the
        for (j=1;j<=n;j++) hessin[i][j]=0.0;            unit matrix.
        hessin[i][i]=1.0;
        xi[i] = -g[i];                              Initial line direction.
        sum += p[i]*p[i];
    }
    stpmax=STPMX*FMAX(sqrt(sum),(float)n);
```

```
    for (its=1;its<=ITMAX;its++) {              Main loop over the iterations.
        *iter=its;
        lnsrch(n,p,fp,g,xi,pnew,fret,stpmax,&check,func);
        The new function evaluation occurs in lnsrch; save the function value in fp for the
        next line search. It is usually safe to ignore the value of check.
        fp = *fret;
        for (i=1;i<=n;i++) {
            xi[i]=pnew[i]-p[i];                 Update the line direction,
            p[i]=pnew[i];                       and the current point.
        }
        test=0.0;                               Test for convergence on Δx.
        for (i=1;i<=n;i++) {
            temp=fabs(xi[i])/FMAX(fabs(p[i]),1.0);
            if (temp > test) test=temp;
        }
        if (test < TOLX) {
            FREEALL
            return;
        }
        for (i=1;i<=n;i++) dg[i]=g[i];          Save the old gradient,
        (*dfunc)(p,g);                          and get the new gradient.
        test=0.0;                               Test for convergence on zero gradient.
        den=FMAX(*fret,1.0);
        for (i=1;i<=n;i++) {
            temp=fabs(g[i])*FMAX(fabs(p[i]),1.0)/den;
            if (temp > test) test=temp;
        }
        if (test < gtol) {
            FREEALL
            return;
        }
        for (i=1;i<=n;i++) dg[i]=g[i]-dg[i];    Compute difference of gradients,
        for (i=1;i<=n;i++) {                    and difference times current matrix.
            hdg[i]=0.0;
            for (j=1;j<=n;j++) hdg[i] += hessin[i][j]*dg[j];
        }
        fac=fae=sumdg=sumxi=0.0;                Calculate dot products for the denomi-
        for (i=1;i<=n;i++) {                        nators.
            fac += dg[i]*xi[i];
            fae += dg[i]*hdg[i];
            sumdg += SQR(dg[i]);
            sumxi += SQR(xi[i]);
        }
        if (fac > sqrt(EPS*sumdg*sumxi)) {      Skip update if fac not sufficiently posi-
            fac=1.0/fac;                            tive.
            fad=1.0/fae;
            The vector that makes BFGS different from DFP:
            for (i=1;i<=n;i++) dg[i]=fac*xi[i]-fad*hdg[i];
            for (i=1;i<=n;i++) {                The BFGS updating formula:
                for (j=i;j<=n;j++) {
                    hessin[i][j] += fac*xi[i]*xi[j]
                    -fad*hdg[i]*hdg[j]+fae*dg[i]*dg[j];
                    hessin[j][i]=hessin[i][j];
                }
            }
        }
        for (i=1;i<=n;i++) {                    Now calculate the next direction to go,
            xi[i]=0.0;
            for (j=1;j<=n;j++) xi[i] -= hessin[i][j]*g[j];
        }
    }                                           and go back for another iteration.
    nrerror("too many iterations in dfpmin");
    FREEALL
}
```

Quasi-Newton methods like `dfpmin` work well with the approximate line minimization done by `lnsrch`. The routines `powell` (§10.5) and `frprmn` (§10.6), however, need more accurate line minimization, which is carried out by the routine `linmin`.

Advanced Implementations of Variable Metric Methods

Although rare, it can conceivably happen that roundoff errors cause the matrix $\mathbf{H}_i$ to become nearly singular or non-positive-definite. This can be serious, because the supposed search directions might then not lead downhill, and because nearly singular $\mathbf{H}_i$'s tend to give subsequent $\mathbf{H}_i$'s that are also nearly singular.

There is a simple fix for this rare problem, the same as was mentioned in §10.4: In case of any doubt, you should *restart* the algorithm at the claimed minimum point, and see if it goes anywhere. Simple, but not very elegant. Modern implementations of variable metric methods deal with the problem in a more sophisticated way.

Instead of building up an approximation to $\mathbf{A}^{-1}$, it is possible to build up an approximation of $\mathbf{A}$ itself. Then, instead of calculating the left-hand side of (10.7.4) directly, one solves the set of linear equations

$$\mathbf{A} \cdot (\mathbf{x} - \mathbf{x}_i) = -\nabla f(\mathbf{x}_i) \tag{10.7.11}$$

At first glance this seems like a bad idea, since solving (10.7.11) is a process of order N^3 — and anyway, how does this help the roundoff problem? The trick is not to store $\mathbf{A}$ but rather a triangular decomposition of $\mathbf{A}$, its *Cholesky decomposition* (cf. §2.9). The updating formula used for the Cholesky decomposition of $\mathbf{A}$ is of order N^2 and can be arranged to guarantee that the matrix remains positive definite and nonsingular, even in the presence of finite roundoff. This method is due to Gill and Murray [1,2].

CITED REFERENCES AND FURTHER READING:

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.1, §§3–6 (by K. W. Brodlie). [2]

Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press), pp. 56ff. [3]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 467–468.

10.8 Linear Programming and the Simplex Method

The subject of *linear programming*, sometimes called *linear optimization*, concerns itself with the following problem: For N independent variables $x_1, \ldots, x_N$, *maximize* the function

$$z = a_{01}x_1 + a_{02}x_2 + \cdots + a_{0N}x_N \tag{10.8.1}$$

subject to the primary constraints

$$x_1 \geq 0, \quad x_2 \geq 0, \quad \ldots \quad x_N \geq 0 \tag{10.8.2}$$

and simultaneously subject to $M = m_1 + m_2 + m_3$ additional constraints, m_1 of them of the form

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{iN}x_N \le b_i \qquad (b_i \ge 0) \qquad i = 1, \ldots, m_1 \tag{10.8.3}$$

m_2 of them of the form

$$a_{j1}x_1 + a_{j2}x_2 + \cdots + a_{jN}x_N \ge b_j \ge 0 \qquad j = m_1 + 1, \ldots, m_1 + m_2 \tag{10.8.4}$$

and m_3 of them of the form

$$\begin{aligned} a_{k1}x_1 + a_{k2}x_2 + \cdots + a_{kN}x_N = b_k \ge 0 \\ k = m_1 + m_2 + 1, \ldots, m_1 + m_2 + m_3 \end{aligned} \tag{10.8.5}$$

The various a_{ij}'s can have either sign, or be zero. The fact that the b's must all be nonnegative (as indicated by the final inequality in the above three equations) is a matter of convention only, since you can multiply any contrary inequality by -1. There is no particular significance in the number of constraints M being less than, equal to, or greater than the number of unknowns N.

A set of values $x_1 \ldots x_N$ that satisfies the constraints (10.8.2)–(10.8.5) is called a *feasible vector*. The function that we are trying to maximize is called the *objective function*. The feasible vector that maximizes the objective function is called the *optimal feasible vector*. An optimal feasible vector can fail to exist for two distinct reasons: (i) there are *no* feasible vectors, i.e., the given constraints are incompatible, or (ii) there is no maximum, i.e., there is a direction in N space where one or more of the variables can be taken to infinity while still satisfying the constraints, giving an unbounded value for the objective function.

As you see, the subject of linear programming is surrounded by notational and terminological thickets. Both of these thorny defenses are lovingly cultivated by a coterie of stern acolytes who have devoted themselves to the field. Actually, the basic ideas of linear programming are quite simple. Avoiding the shrubbery, we want to teach you the basics by means of a couple of specific examples; it should then be quite obvious how to generalize.

Why is linear programming so important? (i) Because "nonnegativity" is the usual constraint on any variable x_i that represents the tangible amount of some physical commodity, like guns, butter, dollars, units of vitamin E, food calories, kilowatt hours, mass, etc. Hence equation (10.8.2). (ii) Because one is often interested in additive (linear) limitations or bounds imposed by man or nature: minimum nutritional requirement, maximum affordable cost, maximum on available labor or capital, minimum tolerable level of voter approval, etc. Hence equations (10.8.3)–(10.8.5). (iii) Because the function that one wants to optimize may be linear, or else may at least be approximated by a linear function — since that is the problem that linear programming *can* solve. Hence equation (10.8.1). For a short, semipopular survey of linear programming applications, see Bland [1].

Here is a specific example of a problem in linear programming, which has $N = 4$, $m_1 = 2$, $m_2 = m_3 = 1$, hence $M = 4$:

$$\text{Maximize} \quad z = x_1 + x_2 + 3x_3 - \tfrac{1}{2}x_4 \tag{10.8.6}$$

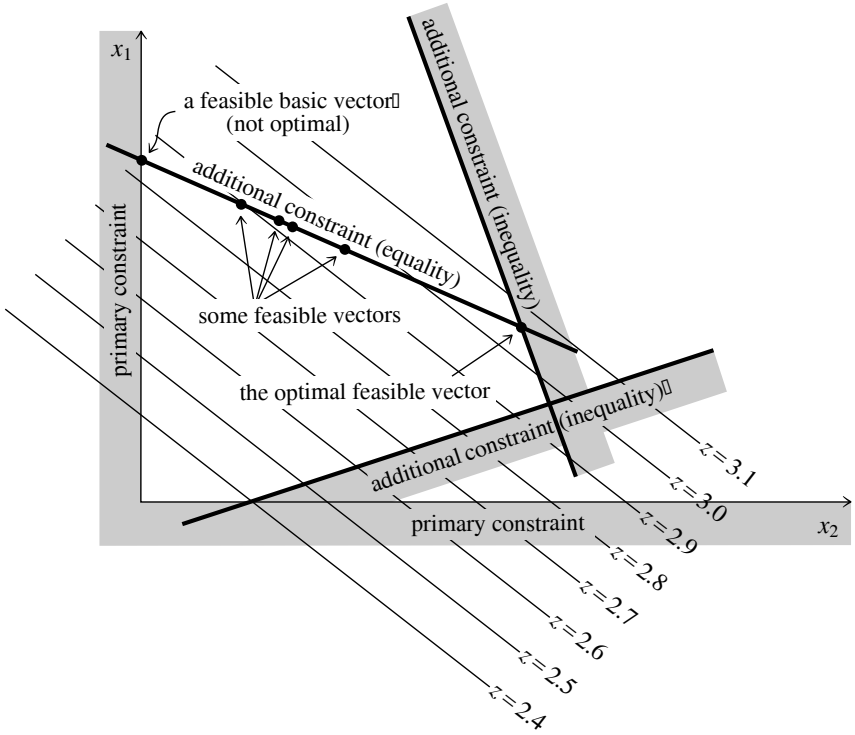


Figure 10.8.1. Basic concepts of linear programming. The case of only two independent variables, x_1, x_2, is shown. The linear function z, to be maximized, is represented by its contour lines. Primary constraints require x_1 and x_2 to be positive. Additional constraints may restrict the solution to regions (inequality constraints) or to surfaces of lower dimensionality (equality constraints). Feasible vectors satisfy all constraints. Feasible basic vectors also lie on the boundary of the allowed region. The simplex method steps among feasible basic vectors until the optimal feasible vector is found.

with all the x's nonnegative and also with

$$\begin{aligned} x_1 + 2x_3 &\le 740 \\ 2x_2 - 7x_4 &\le 0 \\ x_2 - x_3 + 2x_4 &\ge \tfrac{1}{2} \\ x_1 + x_2 + x_3 + x_4 &= 9 \end{aligned} \tag{10.8.7}$$

The answer turns out to be (to 2 decimals) $x_1 = 0$, $x_2 = 3.33$, $x_3 = 4.73$, $x_4 = 0.95$. In the rest of this section we will learn how this answer is obtained. Figure 10.8.1 summarizes some of the terminology thus far.

Fundamental Theorem of Linear Optimization

Imagine that we start with a full N-dimensional space of candidate vectors. Then (in mind's eye, at least) we carve away the regions that are eliminated in turn by each imposed constraint. Since the constraints are linear, every boundary introduced by this process is a plane, or rather hyperplane. Equality constraints of the form (10.8.5)

force the feasible region onto hyperplanes of smaller dimension, while inequalities simply divide the then-feasible region into allowed and nonallowed pieces.

When all the constraints are imposed, either we are left with some feasible region or else there are no feasible vectors. Since the feasible region is bounded by hyperplanes, it is geometrically a kind of convex polyhedron or simplex (cf. §10.4). If there is a feasible region, can the optimal feasible vector be somewhere in its interior, away from the boundaries? No, because the objective function is linear. This means that it always has a nonzero vector gradient. This, in turn, means that we could always increase the objective function by running up the gradient until we hit a boundary wall.

The boundary of any geometrical region has one less dimension than its interior. Therefore, we can now run up the gradient projected into the boundary wall until we reach an edge of that wall. We can then run up that edge, and so on, down through whatever number of dimensions, until we finally arrive at a point, a *vertex* of the original simplex. Since this point has all N of its coordinates defined, it must be the solution of N simultaneous *equalities* drawn from the original set of equalities and inequalities (10.8.2)–(10.8.5).

Points that are feasible vectors and that satisfy N of the original constraints as equalities, are termed *feasible basic vectors*. If $N > M$, then a feasible basic vector has *at least* $N - M$ of its components equal to zero, since at least that many of the constraints (10.8.2) will be needed to make up the total of N. Put the other way, *at most* M components of a feasible basic vector are nonzero. In the example (10.8.6)–(10.8.7), you can check that the solution as given satisfies as equalities the last three constraints of (10.8.7) and the constraint $x_1 \geq 0$, for the required total of 4.

Put together the two preceding paragraphs and you have the *Fundamental Theorem of Linear Optimization*: If an optimal feasible vector exists, then there is a feasible basic vector that is optimal. (Didn't we warn you about the terminological thicket?)

The importance of the fundamental theorem is that it reduces the optimization problem to a "combinatorial" problem, that of determining which N constraints (out of the $M + N$ constraints in 10.8.2–10.8.5) should be satisfied by the optimal feasible vector. We have only to keep trying different combinations, and computing the objective function for each trial, until we find the best.

Doing this blindly would take halfway to forever. The *simplex method*, first published by Dantzig in 1948 (see [2]), is a way of organizing the procedure so that (i) a series of combinations is tried for which the objective function increases at each step, and (ii) the optimal feasible vector is reached after a number of iterations that is almost always no larger than of order M or N, whichever is larger. An interesting mathematical sidelight is that this second property, although known empirically ever since the simplex method was devised, was not proved to be true until the 1982 work of Stephen Smale. (For a contemporary account, see [3].)

Simplex Method for a Restricted Normal Form

A linear programming problem is said to be in *normal form* if it has no constraints in the form (10.8.3) or (10.8.4), but rather only equality constraints of the form (10.8.5) and nonnegativity constraints of the form (10.8.2).

For our purposes it will be useful to consider an even more restricted set of cases, with this additional property: Each equality constraint of the form (10.8.5) must have at least one variable that has a positive coefficient and *that appears uniquely in that one constraint only*. We can then choose one such variable in each constraint equation, and solve that constraint equation for it. The variables thus chosen are called *left-hand variables* or *basic variables*, and there are exactly M $(= m_3)$ of them. The remaining $N - M$ variables are called *right-hand variables* or *nonbasic variables*. Obviously this *restricted normal form* can be achieved only in the case $M \leq N$, so that is the case that we will consider.

You may be thinking that our restricted normal form is so specialized that it is unlikely to include the linear programming problem that you wish to solve. Not at all! We will presently show how *any* linear programming problem can be transformed into restricted normal form. Therefore bear with us and learn how to apply the simplex method to a restricted normal form.

Here is an example of a problem in restricted normal form:

$$\text{Maximize} \quad z = 2x_2 - 4x_3 \tag{10.8.8}$$

with x_1, x_2, x_3, and x_4 all nonnegative and also with

$$\begin{aligned} x_1 &= 2 - 6x_2 + x_3 \\ x_4 &= 8 + 3x_2 - 4x_3 \end{aligned} \tag{10.8.9}$$

This example has $N = 4$, $M = 2$; the left-hand variables are x_1 and x_4; the right-hand variables are x_2 and x_3. The objective function (10.8.8) is written so as to depend only on right-hand variables; note, however, that this is not an actual restriction on objective functions in restricted normal form, since any left-hand variables appearing in the objective function could be eliminated algebraically by use of (10.8.9) or its analogs.

For any problem in restricted normal form, we can instantly read off a feasible basic vector (although not necessarily the *optimal* feasible basic vector). Simply set all right-hand variables equal to zero, and equation (10.8.9) then gives the values of the left-hand variables for which the constraints are satisfied. The idea of the simplex method is to proceed by a series of exchanges. In each exchange, a right-hand variable and a left-hand variable change places. At each stage we maintain a problem in restricted normal form that is equivalent to the original problem.

It is notationally convenient to record the information content of equations (10.8.8) and (10.8.9) in a so-called *tableau*, as follows:

		x_2	x_3
z	0	2	-4
x_1	2	-6	1
x_4	8	3	-4

(10.8.10)

You should study (10.8.10) to be sure that you understand where each entry comes from, and how to translate back and forth between the tableau and equation formats of a problem in restricted normal form.

The first step in the simplex method is to examine the top row of the tableau, which we will call the "z-row." Look at the entries in columns labeled by right-hand variables (we will call these "right-columns"). We want to imagine in turn the effect of increasing each right-hand variable from its present value of zero, while leaving all the other right-hand variables at zero. Will the objective function increase or decrease? The answer is given by the sign of the entry in the z-row. Since we want to increase the objective function, only right columns having positive z-row entries are of interest. In (10.8.10) there is only one such column, whose z-row entry is 2.

The second step is to examine the column entries below each z-row entry that was selected by step one. We want to ask how much we can increase the right-hand variable before one of the left-hand variables is driven negative, which is not allowed. If the tableau element at the intersection of the right-hand column and the left-hand variable's row is positive, then it poses no restriction: the corresponding left-hand variable will just be driven more and more positive. If *all* the entries in any right-hand column are positive, then there is no bound on the objective function and (having said so) we are done with the problem.

If one or more entries below a positive z-row entry are negative, then we have to figure out which such entry first limits the increase of that column's right-hand variable. Evidently the limiting increase is given by dividing the element in the right-hand column (which is called the *pivot element*) into the element in the "constant column" (leftmost column) of the pivot element's row. A value that is small in magnitude is most restrictive. The increase in the objective function for this choice of pivot element is then that value multiplied by the z-row entry of that column. We repeat this procedure on all possible right-hand columns to find the pivot element with the largest such increase. That completes our "choice of a pivot element."

In the above example, the only positive z-row entry is 2. There is only one negative entry below it, namely -6, so this is the pivot element. Its constant-column entry is 2. This pivot will therefore allow x_2 to be increased by $2 \div |6|$, which results in an increase of the objective function by an amount $(2 \times 2) \div |6|$.

The third step is to *do* the increase of the selected right-hand variable, thus making it a left-hand variable; and simultaneously to modify the left-hand variables, reducing the pivot-row element to zero and thus making it a right-hand variable. For our above example let's do this first by hand: We begin by solving the pivot-row equation for the new left-hand variable x_2 in favor of the old one x_1, namely

$$x_1 = 2 - 6x_2 + x_3 \qquad \rightarrow \qquad x_2 = \tfrac{1}{3} - \tfrac{1}{6}x_1 + \tfrac{1}{6}x_3 \tag{10.8.11}$$

We then substitute this into the old z-row,

$$z = 2x_2 - 4x_3 = 2\left[\tfrac{1}{3} - \tfrac{1}{6}x_1 + \tfrac{1}{6}x_3\right] - 4x_3 = \tfrac{2}{3} - \tfrac{1}{3}x_1 - \tfrac{11}{3}x_3 \tag{10.8.12}$$

and into all other left-variable rows, in this case only x_4,

$$x_4 = 8 + 3\left[\tfrac{1}{3} - \tfrac{1}{6}x_1 + \tfrac{1}{6}x_3\right] - 4x_3 = 9 - \tfrac{1}{2}x_1 - \tfrac{7}{2}x_3 \tag{10.8.13}$$

Equations (10.8.11)–(10.8.13) form the new tableau

		x_1	x_3
z	$\frac{2}{3}$	$-\frac{1}{3}$	$-\frac{11}{3}$
x_2	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{6}$
x_4	9	$-\frac{1}{2}$	$-\frac{7}{2}$

(10.8.14)

The fourth step is to go back and repeat the first step, looking for another possible increase of the objective function. We do this as many times as possible, that is, until all the right-hand entries in the z-row are negative, signaling that no further increase is possible. In the present example, this already occurs in (10.8.14), so we are done.

The answer can now be read from the constant column of the final tableau. In (10.8.14) we see that the objective function is maximized to a value of $2/3$ for the solution vector $x_2 = 1/3$, $x_4 = 9$, $x_1 = x_3 = 0$.

Now look back over the procedure that led from (10.8.10) to (10.8.14). You will find that it could be summarized entirely in tableau format as a series of prescribed elementary matrix operations:

- Locate the pivot element and save it.
- Save the whole pivot column.
- Replace each row, except the pivot row, by that linear combination of itself and the pivot row which makes its pivot-column entry zero.
- Divide the pivot row by the negative of the pivot.
- Replace the pivot element by the reciprocal of its saved value.
- Replace the rest of the pivot column by its saved values divided by the saved pivot element.

This is the sequence of operations actually performed by a linear programming routine, such as the one that we will presently give.

You should now be able to solve almost any linear programming problem that starts in restricted normal form. The only special case that might stump you is if an entry in the constant column turns out to be zero at some stage, so that a left-hand variable is zero at the same time as all the right-hand variables are zero. This is called a *degenerate feasible vector*. To proceed, you may need to exchange the degenerate left-hand variable for one of the right-hand variables, perhaps even making several such exchanges.

Writing the General Problem in Restricted Normal Form

Here is a pleasant surprise. There exist a couple of clever tricks that render trivial the task of translating a general linear programming problem into restricted normal form!

First, we need to get rid of the inequalities of the form (10.8.3) or (10.8.4), for example, the first three constraints in (10.8.7). We do this by adding to the problem so-called *slack variables* which, when their nonnegativity is required, convert the inequalities to equalities. We will denote slack variables as y_i. There will be $m_1 + m_2$ of them. Once they are introduced, you treat them on an equal footing with the original variables x_i; then, at the very end, you simply ignore them.

For example, introducing slack variables leaves (10.8.6) unchanged but turns (10.8.7) into

$$\begin{aligned} x_1 + 2x_3 + y_1 &= 740 \\ 2x_2 - 7x_4 + y_2 &= 0 \\ x_2 - x_3 + 2x_4 - y_3 &= \tfrac{1}{2} \\ x_1 + x_2 + x_3 + x_4 &= 9 \end{aligned} \tag{10.8.15}$$

(Notice how the sign of the coefficient of the slack variable is determined by which sense of inequality it is replacing.)

Second, we need to insure that there is a set of M left-hand vectors, so that we can set up a starting tableau in restricted normal form. (In other words, we need to find a "feasible basic starting vector.") The trick is again to invent new variables! There are M of these, and they are called *artificial variables*; we denote them by z_i. You put exactly one artificial variable into each constraint equation on the following model for the example (10.8.15):

$$\begin{aligned} z_1 &= 740 - x_1 - 2x_3 - y_1 \\ z_2 &= -2x_2 + 7x_4 - y_2 \\ z_3 &= \tfrac{1}{2} - x_2 + x_3 - 2x_4 + y_3 \\ z_4 &= 9 - x_1 - x_2 - x_3 - x_4 \end{aligned} \tag{10.8.16}$$

Our example is now in restricted normal form.

Now you may object that (10.8.16) is not the same problem as (10.8.15) or (10.8.7) *unless all the z_i's are zero*. Right you are! There is some subtlety here! We must proceed to solve our problem in two phases. First phase: We replace our objective function (10.8.6) by a so-called *auxiliary objective function*

$$z' \equiv -z_1 - z_2 - z_3 - z_4 = -(749\tfrac{1}{2} - 2x_1 - 4x_2 - 2x_3 + 4x_4 - y_1 - y_2 + y_3) \tag{10.8.17}$$

(where the last equality follows from using 10.8.16). We now perform the simplex method on the auxiliary objective function (10.8.17) with the constraints (10.8.16). Obviously the auxiliary objective function will be maximized for nonnegative z_i's if all the z_i's are zero. We therefore expect the simplex method in this first phase to produce a set of left-hand variables drawn from the x_i's and y_i's only, with all the z_i's being right-hand variables. Aha! We then cross out the z_i's, leaving a problem involving only x_i's and y_i's in restricted normal form. In other words, the first phase produces an initial feasible basic vector. Second phase: Solve the problem produced by the first phase, using the original objective function, not the auxiliary.

And what if the first phase *doesn't* produce zero values for all the z_i's? That signals that there is *no* initial feasible basic vector, i.e., that the constraints given to us are inconsistent among themselves. Report that fact, and you are done.

Here is how to translate into tableau format the information needed for both the first and second phases of the overall method. As before, the underlying problem

to be solved is as posed in equations (10.8.6)–(10.8.7).

		x_1	x_2	x_3	x_4	y_1	y_2	y_3
z	0	1	1	3	$-\frac{1}{2}$	0	0	0
z_1	740	-1	0	-2	0	-1	0	0
z_2	0	0	-2	0	7	0	-1	0
z_3	$\frac{1}{2}$	0	-1	1	-2	0	0	1
z_4	9	-1	-1	-1	-1	0	0	0
z'	$-749\frac{1}{2}$	2	4	2	-4	1	1	-1

(10.8.18)

This is not as daunting as it may, at first sight, appear. The table entries inside the box of double lines are no more than the coefficients of the original problem (10.8.6)–(10.8.7) organized into a tabular form. In fact, these entries, along with the values of N, M, m_1, m_2, and m_3, are the only input that is needed by the simplex method routine below. The columns under the slack variables y_i simply record whether each of the M constraints is of the form $\leq$, $\geq$, or $=$; this is redundant information with the values m_1, m_2, m_3, as long as we are sure to enter the rows of the tableau in the correct respective order. The coefficients of the auxiliary objective function (bottom row) are just the negatives of the column sums of the rows above, so these are easily calculated automatically.

The output from a simplex routine will be (i) a flag telling whether a finite solution, no solution, or an unbounded solution was found, and (ii) an updated tableau. The output tableau that derives from (10.8.18), given to two significant figures, is

		x_1	y_2	y_3	$\cdots$
z	17.03	$-.95$	$-.05$	-1.05	$\cdots$
x_2	3.33	$-.35$	$-.15$	.35	$\cdots$
x_3	4.73	$-.55$	.05	$-.45$	$\cdots$
x_4	.95	$-.10$	.10	.10	$\cdots$
y_1	730.55	.10	$-.10$	.90	$\cdots$

(10.8.19)

A little counting of the x_i's and y_i's will convince you that there are $M + 1$ rows (including the z-row) in both the input and the output tableaux, but that only $N + 1 - m_3$ columns of the output tableau (including the constant column) contain any useful information, the other columns belonging to now-discarded artificial variables. In the output, the first numerical column contains the solution vector, along with the maximum value of the objective function. Where a slack variable (y_i) appears on the left, the corresponding value is the amount by which its inequality is safely satisfied. Variables that are not left-hand variables in the output tableau have zero values. Slack variables with zero values represent constraints that are satisfied as equalities.

Routine Implementing the Simplex Method

The following routine is based algorithmically on the implementation of Kuenzi, Tzschach, and Zehnder [4]. Aside from input values of M, N, m_1, m_2, m_3, the principal input to the routine is a two-dimensional array a containing the portion of the tableau (10.8.18) that is contained between the double lines. This input occupies the $M + 1$ rows and $N + 1$ columns of a[1..m+1][1..n+1]. Note, however, that reference is made internally to row $M + 2$ of a (used for the auxiliary objective function, just as in 10.8.18). Therefore the variable declared as float **a, must point to allocated memory allowing references in the subrange

$$\texttt{a}[i][k], \quad i = 1 \ldots \texttt{m+2},\ k = 1 \ldots \texttt{n+1} \tag{10.8.20}$$

You will suffer endless agonies if you fail to understand this simple point. Also do not neglect to order the rows of a in the same order as equations (10.8.1), (10.8.3), (10.8.4), and (10.8.5), that is, objective function, $\leq$-constraints, $\geq$-constraints, $=$-constraints.

On output, the tableau a is indexed by two returned arrays of integers. iposv[j] contains, for j$= 1 \ldots M$, the number i whose original variable x_i is now represented by row j+1 of a. These are thus the left-hand variables in the solution. (The first row of a is of course the z-row.) A value $i > N$ indicates that the variable is a y_i rather than an x_i, $x_{N+j} \equiv y_j$. Likewise, izrov[j] contains, for j$= 1 \ldots N$, the number i whose original variable x_i is now a right-hand variable, represented by column j+1 of a. These variables are all zero in the solution. The meaning of $i > N$ is the same as above, except that $i > N + m_1 + m_2$ denotes an artificial or slack variable which was used only internally and should now be entirely ignored.

The flag icase is set to zero if a finite solution is found, $+1$ if the objective function is unbounded, -1 if no solution satisfies the given constraints.

The routine treats the case of degenerate feasible vectors, so don't worry about them. You may also wish to admire the fact that the routine does not require storage for the columns of the tableau (10.8.18) that are to the right of the double line; it keeps track of slack variables by more efficient bookkeeping.

Please note that, as given, the routine is only "semi-sophisticated" in its tests for convergence. While the routine properly implements tests for inequality with zero as tests against some small parameter EPS, it does not adjust this parameter to reflect the scale of the input data. This is adequate for many problems, where the input data do not differ from unity by too many orders of magnitude. If, however, you encounter endless cycling, then you should modify EPS in the routines simplx and simp2. Permuting your variables can also help. Finally, consult [5].

```
#include "nrutil.h"
#define EPS 1.0e-6
Here EPS is the absolute precision, which should be adjusted to the scale of your variables.
#define FREEALL free_ivector(l3,1,m);free_ivector(l1,1,n+1);

void simplx(float **a, int m, int n, int m1, int m2, int m3, int *icase,
    int izrov[], int iposv[])
Simplex method for linear programming. Input parameters a, m, n, mp, np, m1, m2, and m3,
and output parameters a, icase, izrov, and iposv are described above.
{
    void simp1(float **a, int mm, int ll[], int nll, int iabf, int *kp,
```

```
    float *bmax);
void simp2(float **a, int m, int n, int *ip, int kp);
void simp3(float **a, int i1, int k1, int ip, int kp);
int i,ip,is,k,kh,kp,nl1;
int *l1,*l3;
float q1,bmax;

if (m != (m1+m2+m3)) nrerror("Bad input constraint counts in simplx");
l1=ivector(1,n+1);
l3=ivector(1,m);
nl1=n;
for (k=1;k<=n;k++) l1[k]=izrov[k]=k;
```

Initialize index list of columns admissible for exchange, and make all variables initially right-hand.

```
for (i=1;i<=m;i++) {
    if (a[i+1][1] < 0.0) nrerror("Bad input tableau in simplx");
```

Constants b_i must be nonnegative.

```
    iposv[i]=n+i;
```

Initial left-hand variables. m1 type constraints are represented by having their slack variable initially left-hand, with no artificial variable. m2 type constraints have their slack variable initially left-hand, with a minus sign, and their artificial variable handled implicitly during their first exchange. m3 type constraints have their artificial variable initially left-hand.

```
}
if (m2+m3) {                                    Origin is not a feasible starting so-
    for (i=1;i<=m2;i++) l3[i]=1;                    lution: we must do phase one.
```

Initialize list of m2 constraints whose slack variables have never been exchanged out of the initial basis.

```
    for (k=1;k<=(n+1);k++) {                    Compute the auxiliary objective func-
        q1=0.0;                                     tion.
        for (i=m1+1;i<=m;i++) q1 += a[i+1][k];
        a[m+2][k] = -q1;
    }
    for (;;) {
        simp1(a,m+1,l1,nl1,0,&kp,&bmax);        Find max. coeff. of auxiliary objec-
        if (bmax <= EPS && a[m+2][1] < -EPS) {      tive fn.
            *icase = -1;
```

Auxiliary objective function is still negative and can't be improved, hence no feasible solution exists.

```
            FREEALL return;
        } else if (bmax <= EPS && a[m+2][1] <= EPS) {
```

Auxiliary objective function is zero and can't be improved; we have a feasible starting vector. Clean out the artificial variables corresponding to any remaining equality constraints by goto one and then move on to phase two.

```
            for (ip=m1+m2+1;ip<=m;ip++) {
                if (iposv[ip] == (ip+n)) {      Found an artificial variable for an
                    simp1(a,ip,l1,nl1,1,&kp,&bmax);     equality constraint.
                    if (bmax > EPS)             Exchange with column correspond-
                        goto one;                   ing to maximum pivot element
                }                                   in row.
            }
            for (i=m1+1;i<=m1+m2;i++)           Change sign of row for any m2 con-
                if (l3[i-m1] == 1)                  straints still present from the ini-
                    for (k=1;k<=n+1;k++)            tial basis.
                        a[i+1][k] = -a[i+1][k];
            break;                              Go to phase two.
        }
        simp2(a,m,n,&ip,kp);                    Locate a pivot element (phase one).
        if (ip == 0) {                          Maximum of auxiliary objective func-
            *icase = -1;                            tion is unbounded, so no feasi-
            FREEALL return;                         ble solution exists.
        }
one:    simp3(a,m+1,n,ip,kp);
```

Exchange a left- and a right-hand variable (phase one), then update lists.

```
            if (iposv[ip] >= (n+m1+m2+1)) {        Exchanged out an artificial variable
                for (k=1;k<=nl1;k++)                   for an equality constraint. Make
                    if (l1[k] == kp) break;            sure it stays out by removing it
                --nl1;                                 from the l1 list.
                for (is=k;is<=nl1;is++) l1[is]=l1[is+1];
            } else {
                kh=iposv[ip]-m1-n;
                if (kh >= 1 && l3[kh]) {           Exchanged out an m2 type constraint
                    l3[kh]=0;                          for the first time. Correct the
                    ++a[m+2][kp+1];                    pivot column for the minus sign
                    for (i=1;i<=m+2;i++)               and the implicit artificial vari-
                        a[i][kp+1] = -a[i][kp+1];      able.
                }
            }
            is=izrov[kp];                          Update lists of left- and right-hand
            izrov[kp]=iposv[ip];                       variables.
            iposv[ip]=is;
        }                                          Still in phase one, go back to the
    }                                                  for(;;).
    End of phase one code for finding an initial feasible solution. Now, in phase two, optimize
    it.
    for (;;) {
        simp1(a,0,l1,nl1,0,&kp,&bmax);             Test the z-row for doneness.
        if (bmax <= EPS) {                         Done. Solution found. Return with
            *icase=0;                                  the good news.
            FREEALL return;
        }
        simp2(a,m,n,&ip,kp);                       Locate a pivot element (phase two).
        if (ip == 0) {                             Objective function is unbounded. Re-
            *icase=1;                                  port and return.
            FREEALL return;
        }
        simp3(a,m,n,ip,kp);                        Exchange a left- and a right-hand
        is=izrov[kp];                                  variable (phase two),
        izrov[kp]=iposv[ip];
        iposv[ip]=is;
    }                                              and return for another iteration.
}
```

The preceding routine makes use of the following utility functions.

```
#include <math.h>

void simp1(float **a, int mm, int ll[], int nll, int iabf, int *kp,
    float *bmax)
Determines the maximum of those elements whose index is contained in the supplied list ll,
either with or without taking the absolute value, as flagged by iabf.
{
    int k;
    float test;

    if (nll <= 0)                  No eligible columns.
        *bmax=0.0;
    else {
        *kp=ll[1];
        *bmax=a[mm+1][*kp+1];
        for (k=2;k<=nll;k++) {
            if (iabf == 0)
                test=a[mm+1][ll[k]+1]-(*bmax);
```

```
            else
                test=fabs(a[mm+1][ll[k]+1])-fabs(*bmax);
            if (test > 0.0) {
                *bmax=a[mm+1][ll[k]+1];
                *kp=ll[k];
            }
        }
    }
}
```

```
#define EPS 1.0e-6

void simp2(float **a, int m, int n, int *ip, int kp)
Locate a pivot element, taking degeneracy into account.
{
    int k,i;
    float qp,q0,q,q1;

    *ip=0;
    for (i=1;i<=m;i++)
        if (a[i+1][kp+1] < -EPS) break;        Any possible pivots?
    if (i>m) return;
    q1 = -a[i+1][1]/a[i+1][kp+1];
    *ip=i;
    for (i=*ip+1;i<=m;i++) {
        if (a[i+1][kp+1] < -EPS) {
            q = -a[i+1][1]/a[i+1][kp+1];
            if (q < q1) {
                *ip=i;
                q1=q;
            } else if (q == q1) {              We have a degeneracy.
                for (k=1;k<=n;k++) {
                    qp = -a[*ip+1][k+1]/a[*ip+1][kp+1];
                    q0 = -a[i+1][k+1]/a[i+1][kp+1];
                    if (q0 != qp) break;
                }
                if (q0 < qp) *ip=i;
            }
        }
    }
}
```

```
void simp3(float **a, int i1, int k1, int ip, int kp)
Matrix operations to exchange a left-hand and right-hand variable (see text).
{
    int kk,ii;
    float piv;

    piv=1.0/a[ip+1][kp+1];
    for (ii=1;ii<=i1+1;ii++)
        if (ii-1 != ip) {
            a[ii][kp+1] *= piv;
            for (kk=1;kk<=k1+1;kk++)
                if (kk-1 != kp)
                    a[ii][kk] -= a[ip+1][kk]*a[ii][kp+1];
        }
```

```
    for (kk=1;kk<=k1+1;kk++)
        if (kk-1 != kp) a[ip+1][kk] *= -piv;
    a[ip+1][kp+1]=piv;
}
```

Other Topics Briefly Mentioned

Every linear programming problem in normal form with N variables and M constraints has a corresponding *dual* problem with M variables and N constraints. The tableau of the dual problem is, in essence, the transpose of the tableau of the original (sometimes called *primal*) problem. It is possible to go from a solution of the dual to a solution of the primal. This can occasionally be computationally useful, but generally it is no big deal.

The *revised simplex method* is exactly equivalent to the simplex method in its choice of which left-hand and right-hand variables are exchanged. Its computational effort is not significantly less than that of the simplex method. It does differ in the organization of its storage, requiring only a matrix of size $M \times M$, rather than $M \times N$, in its intermediate stages. If you have a lot of constraints, and memory size is one of them, then you should look into it.

The *primal-dual algorithm* and the *composite simplex algorithm* are two different methods for avoiding the two phases of the usual simplex method: Progress is made simultaneously towards finding a feasible solution and finding an optimal solution. There seems to be no clearcut evidence that these methods are superior to the usual method by any factor substantially larger than the "tender-loving-care factor" (which reflects the programming effort of the proponents).

Problems where the objective function and/or one or more of the constraints are replaced by expressions nonlinear in the variables are called *nonlinear programming problems*. The literature on such problems is vast, but outside our scope. The special case of quadratic expressions is called *quadratic programming*. Optimization problems where the variables take on only integer values are called *integer programming* problems, a special case of *discrete optimization* generally. The next section looks at a particular kind of discrete optimization problem.

CITED REFERENCES AND FURTHER READING:

Bland, R.G. 1981, *Scientific American*, vol. 244 (June), pp. 126–144. [1]

Dantzig, G.B. 1963, *Linear Programming and Extensions* (Princeton, NJ: Princeton University Press). [2]

Kolata, G. 1982, *Science*, vol. 217, p. 39. [3]

Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley), Chapters 7–8.

Cooper, L., and Steinberg, D. 1970, *Introduction to Methods of Optimization* (Philadelphia: Saunders).

Gass, S.T. 1969, *Linear Programming*, 3rd ed. (New York: McGraw-Hill).

Murty, K.G. 1976, *Linear and Combinatorial Programming* (New York: Wiley).

Land, A.H., and Powell, S. 1973, *Fortran Codes for Mathematical Programming* (London: Wiley-Interscience).

Kuenzi, H.P., Tzschach, H.G., and Zehnder, C.A. 1971, *Numerical Methods of Mathematical Optimization* (New York: Academic Press). [4]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §4.10.

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [5]

10.9 Simulated Annealing Methods

The *method of simulated annealing* [1,2] is a technique that has attracted significant attention as suitable for optimization problems of large scale, especially ones where a desired global extremum is hidden among many, poorer, local extrema. For practical purposes, simulated annealing has effectively "solved" the famous *traveling salesman problem* of finding the shortest cyclical itinerary for a traveling salesman who must visit each of N cities in turn. (Other practical methods have also been found.) The method has also been used successfully for designing complex integrated circuits: The arrangement of several hundred thousand circuit elements on a tiny silicon substrate is optimized so as to minimize interference among their connecting wires [3,4]. Surprisingly, the implementation of the algorithm is relatively simple.

Notice that the two applications cited are both examples of *combinatorial minimization*. There is an objective function to be minimized, as usual; but the space over which that function is defined is not simply the N-dimensional space of N continuously variable parameters. Rather, it is a discrete, but very large, configuration space, like the set of possible orders of cities, or the set of possible allocations of silicon "real estate" blocks to circuit elements. The number of elements in the configuration space is factorially large, so that they cannot be explored exhaustively. Furthermore, since the set is discrete, we are deprived of any notion of "continuing downhill in a favorable direction." The concept of "direction" may not have any meaning in the configuration space.

Below, we will also discuss how to use simulated annealing methods for spaces with continuous control parameters, like those of §§10.4–10.7. This application is actually more complicated than the combinatorial one, since the familiar problem of "long, narrow valleys" again asserts itself. Simulated annealing, as we will see, tries "random" steps; but in a long, narrow valley, almost all random steps are uphill! Some additional finesse is therefore required.

At the heart of the method of simulated annealing is an analogy with thermodynamics, specifically with the way that liquids freeze and crystallize, or metals cool and anneal. At high temperatures, the molecules of a liquid move freely with respect to one another. If the liquid is cooled slowly, thermal mobility is lost. The atoms are often able to line themselves up and form a pure crystal that is completely ordered over a distance up to billions of times the size of an individual atom in all directions. This crystal is the state of minimum energy for this system. The amazing fact is that, for slowly cooled systems, nature is able to find this minimum energy state. In fact, if a liquid metal is cooled quickly or "quenched," it does not reach this state but rather ends up in a polycrystalline or amorphous state having somewhat higher energy.

So the essence of the process is *slow* cooling, allowing ample time for redistribution of the atoms as they lose mobility. This is the technical definition of *annealing*, and it is essential for ensuring that a low energy state will be achieved.

Although the analogy is not perfect, there is a sense in which all of the minimization algorithms thus far in this chapter correspond to rapid cooling or quenching. In all cases, we have gone greedily for the quick, nearby solution: From the starting point, go immediately downhill as far as you can go. This, as often remarked above, leads to a local, but not necessarily a global, minimum. Nature's own minimization algorithm is based on quite a different procedure. The so-called Boltzmann probability distribution,

$$\text{Prob}\,(E) \sim \exp(-E/kT) \tag{10.9.1}$$

expresses the idea that a system in thermal equilibrium at temperature T has its energy probabilistically distributed among all different energy states E. Even at low temperature, there is a chance, albeit very small, of a system being in a high energy state. Therefore, there is a corresponding chance for the system to get out of a local energy minimum in favor of finding a better, more global, one. The quantity k (Boltzmann's constant) is a constant of nature that relates temperature to energy. In other words, the system sometimes goes *uphill* as well as downhill; but the lower the temperature, the less likely is any significant uphill excursion.

In 1953, Metropolis and coworkers [5] first incorporated these kinds of principles into numerical calculations. Offered a succession of options, a simulated thermodynamic system was assumed to change its configuration from energy E_1 to energy E_2 with probability $p = \exp[-(E_2 - E_1)/kT]$. Notice that if $E_2 < E_1$, this probability is greater than unity; in such cases the change is arbitrarily assigned a probability $p = 1$, i.e., the system *always* took such an option. This general scheme, of always taking a downhill step while *sometimes* taking an uphill step, has come to be known as the Metropolis algorithm.

To make use of the Metropolis algorithm for other than thermodynamic systems, one must provide the following elements:

1. A description of possible system configurations.
2. A generator of random changes in the configuration; these changes are the "options" presented to the system.
3. An objective function E (analog of energy) whose minimization is the goal of the procedure.
4. A control parameter T (analog of temperature) and an *annealing schedule* which tells how it is lowered from high to low values, e.g., after how many random changes in configuration is each downward step in T taken, and how large is that step. The meaning of "high" and "low" in this context, and the assignment of a schedule, may require physical insight and/or trial-and-error experiments.

Combinatorial Minimization: The Traveling Salesman

A concrete illustration is provided by the traveling salesman problem. The proverbial seller visits N cities with given positions (x_i, y_i), returning finally to his or her city of origin. Each city is to be visited only once, and the route is to be made as short as possible. This problem belongs to a class known as *NP-complete* problems, whose computation time for an *exact* solution increases with N as $\exp(\text{const.} \times N)$, becoming rapidly prohibitive in cost as N increases. The traveling salesman problem also belongs to a class of minimization problems for which the objective function E

has many local minima. In practical cases, it is often enough to be able to choose from these a minimum which, even if not absolute, cannot be significantly improved upon. The annealing method manages to achieve this, while limiting its calculations to scale as a small power of N.

As a problem in simulated annealing, the traveling salesman problem is handled as follows:

1. *Configuration.* The cities are numbered $i = 1 \ldots N$ and each has coordinates (x_i, y_i). A configuration is a permutation of the number $1 \ldots N$, interpreted as the order in which the cities are visited.

2. *Rearrangements.* An efficient set of moves has been suggested by Lin [6]. The moves consist of two types: (a) A section of path is removed and then replaced with the same cities running in the opposite order; or (b) a section of path is removed and then replaced in between two cities on another, randomly chosen, part of the path.

3. *Objective Function.* In the simplest form of the problem, E is taken just as the total length of journey,

$$E = L \equiv \sum_{i=1}^{N} \sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} \tag{10.9.2}$$

with the convention that point $N + 1$ is identified with point 1. To illustrate the flexibility of the method, however, we can add the following additional wrinkle: Suppose that the salesman has an irrational fear of flying over the Mississippi River. In that case, we would assign each city a parameter μ_i, equal to $+1$ if it is east of the Mississippi, -1 if it is west, and take the objective function to be

$$E = \sum_{i=1}^{N} \left[\sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} + \lambda(\mu_i - \mu_{i+1})^2 \right] \tag{10.9.3}$$

A penalty 4λ is thereby assigned to any river crossing. The algorithm now finds the shortest path that avoids crossings. The relative importance that it assigns to length of path versus river crossings is determined by our choice of λ. Figure 10.9.1 shows the results obtained. Clearly, this technique can be generalized to include many conflicting goals in the minimization.

4. *Annealing schedule.* This requires experimentation. We first generate some random rearrangements, and use them to determine the range of values of ΔE that will be encountered from move to move. Choosing a starting value for the parameter T which is considerably larger than the largest ΔE normally encountered, we proceed downward in multiplicative steps each amounting to a 10 percent decrease in T. We hold each new value of T constant for, say, $100N$ reconfigurations, or for $10N$ successful reconfigurations, whichever comes first. When efforts to reduce E further become sufficiently discouraging, we stop.

The following traveling salesman program, using the Metropolis algorithm, illustrates the main aspects of the simulated annealing technique for combinatorial problems.

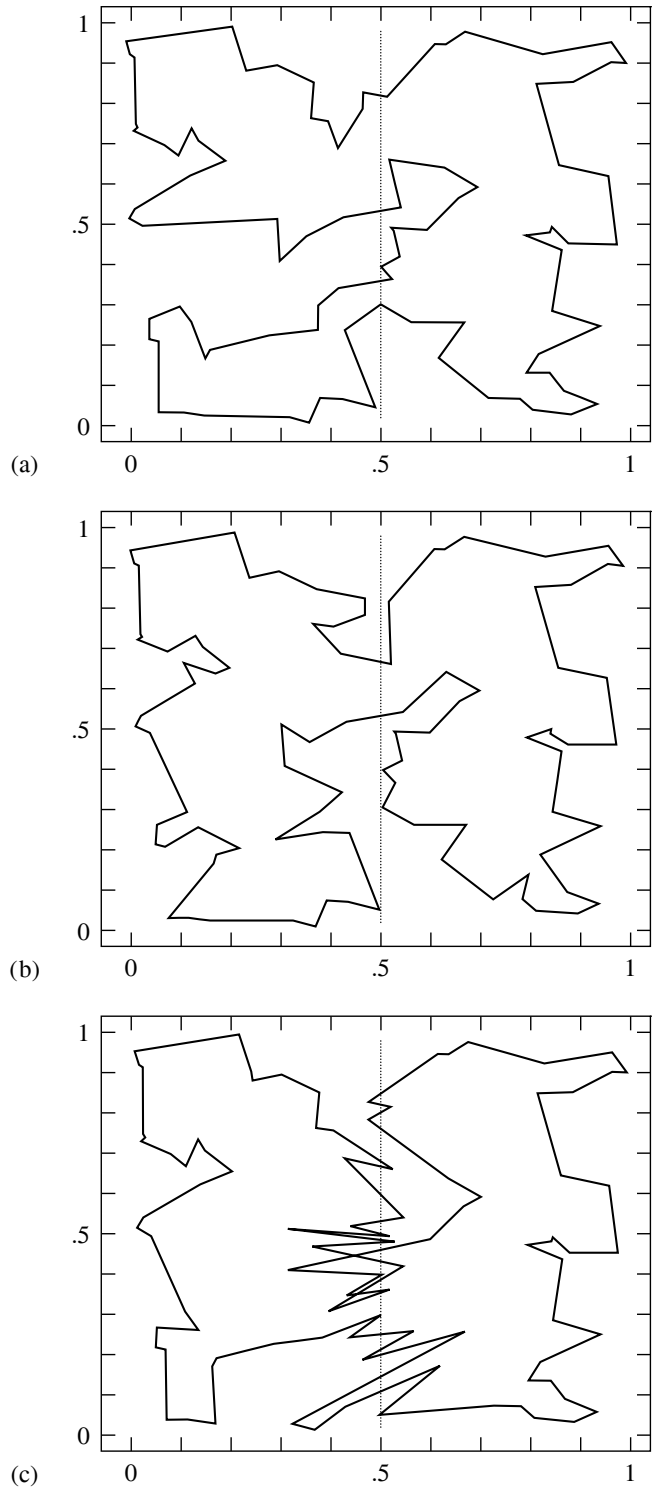


Figure 10.9.1. Traveling salesman problem solved by simulated annealing. The (nearly) shortest path among 100 randomly positioned cities is shown in (a). The dotted line is a river, but there is no penalty in crossing. In (b) the river-crossing penalty is made large, and the solution restricts itself to the minimum number of crossings, two. In (c) the penalty has been made negative: the salesman is actually a smuggler who crosses the river on the flimsiest excuse!

```
#include <stdio.h>
#include <math.h>
#define TFACTR 0.9                  Annealing schedule: reduce t by this factor on each step.
#define ALEN(a,b,c,d) sqrt(((b)-(a))*((b)-(a))+((d)-(c))*((d)-(c)))

void anneal(float x[], float y[], int iorder[], int ncity)
```

This algorithm finds the shortest round-trip path to `ncity` cities whose coordinates are in the arrays `x[1..ncity]`,`y[1..ncity]`. The array `iorder[1..ncity]` specifies the order in which the cities are visited. On input, the elements of `iorder` may be set to any permutation of the numbers 1 to `ncity`. This routine will return the best alternative path it can find.

```
{
    int irbit1(unsigned long *iseed);
    int metrop(float de, float t);
    float ran3(long *idum);
    float revcst(float x[], float y[], int iorder[], int ncity, int n[]);
    void reverse(int iorder[], int ncity, int n[]);
    float trncst(float x[], float y[], int iorder[], int ncity, int n[]);
    void trnspt(int iorder[], int ncity, int n[]);
    int ans,nover,nlimit,i1,i2;
    int i,j,k,nsucc,nn,idec;
    static int n[7];
    long idum;
    unsigned long iseed;
    float path,de,t;

    nover=100*ncity;                Maximum number of paths tried at any temperature.
    nlimit=10*ncity;                Maximum number of successful path changes before con-
    path=0.0;                           tinuing.
    t=0.5;
    for (i=1;i<ncity;i++) {         Calculate initial path length.
        i1=iorder[i];
        i2=iorder[i+1];
        path += ALEN(x[i1],x[i2],y[i1],y[i2]);
    }
    i1=iorder[ncity];               Close the loop by tying path ends together.
    i2=iorder[1];
    path += ALEN(x[i1],x[i2],y[i1],y[i2]);
    idum = -1;
    iseed=111;
    for (j=1;j<=100;j++) {          Try up to 100 temperature steps.
        nsucc=0;
        for (k=1;k<=nover;k++) {
            do {
                n[1]=1+(int) (ncity*ran3(&idum));          Choose beginning of segment
                n[2]=1+(int) ((ncity-1)*ran3(&idum));      ..and end of segment.
                if (n[2] >= n[1]) ++n[2];
                nn=1+((n[1]-n[2]+ncity-1) % ncity);        nn is the number of cities
            } while (nn<3);                                    not on the segment.
            idec=irbit1(&iseed);
            Decide whether to do a segment reversal or transport.
            if (idec == 0) {                           Do a transport.
                n[3]=n[2]+(int) (abs(nn-2)*ran3(&idum))+1;
                n[3]=1+((n[3]-1) % ncity);
                Transport to a location not on the path.
                de=trncst(x,y,iorder,ncity,n);         Calculate cost.
                ans=metrop(de,t);                      Consult the oracle.
                if (ans) {
                    ++nsucc;
                    path += de;
                    trnspt(iorder,ncity,n);            Carry out the transport.
                }
            } else {                                   Do a path reversal.
                de=revcst(x,y,iorder,ncity,n);         Calculate cost.
                ans=metrop(de,t);                      Consult the oracle.
```

```
                if (ans) {
                    ++nsucc;
                    path += de;
                    reverse(iorder,ncity,n);        Carry out the reversal.
                }
            }
            if (nsucc >= nlimit) break;             Finish early if we have enough successful changes.
        }
        printf("\n %s %10.6f %s %12.6f \n","T =",t,
            "     Path Length =",path);
        printf("Successful Moves: %6d\n",nsucc);
        t *= TFACTR;                                Annealing schedule.
        if (nsucc == 0) return;                     If no success, we are done.
    }
}
```

```
#include <math.h>
#define ALEN(a,b,c,d) sqrt(((b)-(a))*((b)-(a))+((d)-(c))*((d)-(c)))
```

`float revcst(float x[], float y[], int iorder[], int ncity, int n[])`
This function returns the value of the cost function for a proposed path reversal. `ncity` is the number of cities, and arrays `x[1..ncity]`,`y[1..ncity]` give the coordinates of these cities. `iorder[1..ncity]` holds the present itinerary. The first two values `n[1]` and `n[2]` of array `n` give the starting and ending cities along the path segment which is to be reversed. On output, `de` is the cost of making the reversal. The actual reversal is not performed by this routine.

```
{
    float xx[5],yy[5],de;
    int j,ii;

    n[3]=1 + ((n[1]+ncity-2) % ncity);      Find the city before n[1] ..
    n[4]=1 + (n[2] % ncity);                .. and the city after n[2].
    for (j=1;j<=4;j++) {
        ii=iorder[n[j]];                    Find coordinates for the four cities involved.
        xx[j]=x[ii];
        yy[j]=y[ii];
    }
    de = -ALEN(xx[1],xx[3],yy[1],yy[3]);    Calculate cost of disconnecting the segment at both ends and reconnecting in the opposite order.
    de -= ALEN(xx[2],xx[4],yy[2],yy[4]);
    de += ALEN(xx[1],xx[4],yy[1],yy[4]);
    de += ALEN(xx[2],xx[3],yy[2],yy[3]);
    return de;
}
```

`void reverse(int iorder[], int ncity, int n[])`
This routine performs a path segment reversal. `iorder[1..ncity]` is an input array giving the present itinerary. The vector `n` has as its first four elements the first and last cities `n[1]`,`n[2]` of the path segment to be reversed, and the two cities `n[3]` and `n[4]` that immediately precede and follow this segment. `n[3]` and `n[4]` are found by function `revcst`. On output, `iorder[1..ncity]` contains the segment from `n[1]` to `n[2]` in reversed order.

```
{
    int nn,j,k,l,itmp;

    nn=(1+((n[2]-n[1]+ncity) % ncity))/2;   This many cities must be swapped to effect the reversal.
    for (j=1;j<=nn;j++) {
        k=1 + ((n[1]+j-2) % ncity);         Start at the ends of the segment and swap pairs of cities, moving toward the center.
        l=1 + ((n[2]-j+ncity) % ncity);
        itmp=iorder[k];
        iorder[k]=iorder[l];
        iorder[l]=itmp;
    }
}
```

```
#include <math.h>
#define ALEN(a,b,c,d) sqrt(((b)-(a))*((b)-(a))+((d)-(c))*((d)-(c)))

float trncst(float x[], float y[], int iorder[], int ncity, int n[])
```

This routine returns the value of the cost function for a proposed path segment transport. `ncity` is the number of cities, and arrays `x[1..ncity]` and `y[1..ncity]` give the city coordinates. `iorder[1..ncity]` is an array giving the present itinerary. The first three elements of array `n` give the starting and ending cities of the path to be transported, and the point among the remaining cities after which it is to be inserted. On output, `de` is the cost of the change. The actual transport is not performed by this routine.

```
{
    float xx[7],yy[7],de;
    int j,ii;

    n[4]=1 + (n[3] % ncity);                    Find the city following n[3]..
    n[5]=1 + ((n[1]+ncity-2) % ncity);          ..and the one preceding n[1]..
    n[6]=1 + (n[2] % ncity);                    ..and the one following n[2].
    for (j=1;j<=6;j++) {
        ii=iorder[n[j]];                        Determine coordinates for the six cities
        xx[j]=x[ii];                                involved.
        yy[j]=y[ii];
    }
    de = -ALEN(xx[2],xx[6],yy[2],yy[6]);        Calculate the cost of disconnecting the
    de -= ALEN(xx[1],xx[5],yy[1],yy[5]);            path segment from n[1] to n[2],
    de -= ALEN(xx[3],xx[4],yy[3],yy[4]);            opening a space between n[3] and
    de += ALEN(xx[1],xx[3],yy[1],yy[3]);            n[4], connecting the segment in the
    de += ALEN(xx[2],xx[4],yy[2],yy[4]);            space, and connecting n[5] to n[6].
    de += ALEN(xx[5],xx[6],yy[5],yy[6]);
    return de;
}
```

```
#include "nrutil.h"

void trnspt(int iorder[], int ncity, int n[])
```

This routine does the actual path transport, once `metrop` has approved. `iorder[1..ncity]` is an input array giving the present itinerary. The array `n` has as its six elements the beginning `n[1]` and end `n[2]` of the path to be transported, the adjacent cities `n[3]` and `n[4]` between which the path is to be placed, and the cities `n[5]` and `n[6]` that precede and follow the path. `n[4]`, `n[5]`, and `n[6]` are calculated by function `trncst`. On output, `iorder` is modified to reflect the movement of the path segment.

```
{
    int m1,m2,m3,nn,j,jj,*jorder;

    jorder=ivector(1,ncity);
    m1=1 + ((n[2]-n[1]+ncity) % ncity);         Find number of cities from n[1] to n[2]
    m2=1 + ((n[5]-n[4]+ncity) % ncity);         ...and the number from n[4] to n[5]
    m3=1 + ((n[3]-n[6]+ncity) % ncity);         ...and the number from n[6] to n[3].
    nn=1;
    for (j=1;j<=m1;j++) {
        jj=1 + ((j+n[1]-2) % ncity);            Copy the chosen segment.
        jorder[nn++]=iorder[jj];
    }
    for (j=1;j<=m2;j++) {                       Then copy the segment from n[4] to
        jj=1+((j+n[4]-2) % ncity);                  n[5].
        jorder[nn++]=iorder[jj];
    }
    for (j=1;j<=m3;j++) {                       Finally, the segment from n[6] to n[3].
        jj=1 + ((j+n[6]-2) % ncity);
        jorder[nn++]=iorder[jj];
    }
    for (j=1;j<=ncity;j++)                      Copy jorder back into iorder.
        iorder[j]=jorder[j];
```

```
    free_ivector(jorder,1,ncity);
}
```

```
#include <math.h>

int metrop(float de, float t)
Metropolis algorithm. metrop returns a boolean variable that issues a verdict on whether
to accept a reconfiguration that leads to a change de in the objective function e. If de<0,
metrop = 1 (true), while if de>0, metrop is only true with probability exp(-de/t), where
t is a temperature determined by the annealing schedule.
{
    float ran3(long *idum);
    static long gljdum=1;

    return de < 0.0 || ran3(&gljdum) < exp(-de/t);
}
```

Continuous Minimization by Simulated Annealing

The basic ideas of simulated annealing are also applicable to optimization problems with continuous N-dimensional control spaces, e.g., finding the (ideally, global) minimum of some function $f(\mathbf{x})$, in the presence of many local minima, where $\mathbf{x}$ is an N-dimensional vector. The four elements required by the Metropolis procedure are now as follows: The value of f is the objective function. The system state is the point $\mathbf{x}$. The control parameter T is, as before, something like a temperature, with an annealing schedule by which it is gradually reduced. And there must be a generator of random changes in the configuration, that is, a procedure for taking a random step from $\mathbf{x}$ to $\mathbf{x} + \Delta\mathbf{x}$.

The last of these elements is the most problematical. The literature to date [7-10] describes several different schemes for choosing $\Delta\mathbf{x}$, none of which, in our view, inspire complete confidence. The problem is one of efficiency: A generator of random changes is inefficient if, *when local downhill moves exist*, it nevertheless almost always proposes an uphill move. A good generator, we think, should not become inefficient in narrow valleys; nor should it become more and more inefficient as convergence to a minimum is approached. Except possibly for [7], all of the schemes that we have seen are inefficient in one or both of these situations.

Our own way of doing simulated annealing minimization on continuous control spaces is to use a modification of the downhill simplex method (§10.4). This amounts to replacing the single point $\mathbf{x}$ as a description of the system state by a simplex of $N + 1$ points. The "moves" are the same as described in §10.4, namely reflections, expansions, and contractions of the simplex. The implementation of the Metropolis procedure is slightly subtle: We *add* a positive, logarithmically distributed random variable, proportional to the temperature T, to the stored function value associated with every vertex of the simplex, and we *subtract* a similar random variable from the function value of every new point that is tried as a replacement point. Like the ordinary Metropolis procedure, this method always accepts a true downhill step, but

sometimes accepts an uphill one. In the limit $T \to 0$, this algorithm reduces exactly to the downhill simplex method and converges to a local minimum.

At a finite value of T, the simplex expands to a scale that approximates the size of the region that can be reached at this temperature, and then executes a stochastic, tumbling Brownian motion within that region, sampling new, approximately random, points as it does so. The efficiency with which a region is explored is independent of its narrowness (for an ellipsoidal valley, the ratio of its principal axes) and orientation. If the temperature is reduced sufficiently slowly, it becomes highly likely that the simplex will shrink into that region containing the lowest relative minimum encountered.

As in all applications of simulated annealing, there can be quite a lot of problem-dependent subtlety in the phrase "sufficiently slowly"; success or failure is quite often determined by the choice of annealing schedule. Here are some possibilities worth trying:

- Reduce T to $(1 - \epsilon)T$ after every m moves, where ϵ/m is determined by experiment.
- Budget a total of K moves, and reduce T after every m moves to a value $T = T_0(1 - k/K)^\alpha$, where k is the cumulative number of moves thus far, and α is a constant, say 1, 2, or 4. The optimal value for α depends on the statistical distribution of relative minima of various depths. Larger values of α spend more iterations at lower temperature.
- After every m moves, set T to β times $f_1 - f_b$, where β is an experimentally determined constant of order 1, f_1 is the smallest function value currently represented in the simplex, and f_b is the best function ever encountered. However, never reduce T by more than some fraction γ at a time.

Another strategic question is whether to do an occasional *restart*, where a vertex of the simplex is discarded in favor of the "best-ever" point. (You must be sure that the best-ever point is not currently in the simplex when you do this!) We have found problems for which restarts — every time the temperature has decreased by a factor of 3, say — are highly beneficial; we have found other problems for which restarts have no positive, or a somewhat negative, effect.

You should compare the following routine, `amebsa`, with its counterpart `amoeba` in §10.4. Note that the argument `iter` is used in a somewhat different manner.

```
#include <math.h>
#include "nrutil.h"
#define GET_PSUM \
                  for (n=1;n<=ndim;n++) {\
                  for (sum=0.0,m=1;m<=mpts;m++) sum += p[m][n];\
                  psum[n]=sum;}
extern long idum;                          Defined and initialized in main.
float tt;                                  Communicates with amotsa.

void amebsa(float **p, float y[], int ndim, float pb[], float *yb, float ftol,
    float (*funk)(float []), int *iter, float temptr)
```

Multidimensional minimization of the function `funk(x)` where `x[1..ndim]` is a vector in `ndim` dimensions, by simulated annealing combined with the downhill simplex method of Nelder and Mead. The input matrix `p[1..ndim+1][1..ndim]` has `ndim+1` rows, each an `ndim`-dimensional vector which is a vertex of the starting simplex. Also input are the following: 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`; `ftol`, the fractional convergence tolerance to be achieved in the function value for an early return; `iter`, and `temptr`. The routine makes `iter` function evaluations at an annealing temperature `temptr`, then returns. You should then de-

crease `temptr` according to your annealing schedule, reset `iter`, and call the routine again (leaving other arguments unaltered between calls). If `iter` is returned with a positive value, then early convergence and return occurred. If you initialize `yb` to a very large value on the first call, then `yb` and `pb[1..ndim]` will subsequently return the best function value and point ever encountered (even if it is no longer a point in the simplex).

```
{
    float amotsa(float **p, float y[], float psum[], int ndim, float pb[],
        float *yb, float (*funk)(float []), int ihi, float *yhi, float fac);
    float ran1(long *idum);
    int i,ihi,ilo,j,m,n,mpts=ndim+1;
    float rtol,sum,swap,yhi,ylo,ynhi,ysave,yt,ytry,*psum;

    psum=vector(1,ndim);
    tt = -temptr;
    GET_PSUM
    for (;;) {
        ilo=1;                                  Determine which point is the highest (worst),
        ihi=2;                                      next-highest, and lowest (best).
        ynhi=ylo=y[1]+tt*log(ran1(&idum));      Whenever we "look at" a vertex, it gets
        yhi=y[2]+tt*log(ran1(&idum));               a random thermal fluctuation.
        if (ylo > yhi) {
            ihi=1;
            ilo=2;
            ynhi=yhi;
            yhi=ylo;
            ylo=ynhi;
        }
        for (i=3;i<=mpts;i++) {                 Loop over the points in the simplex.
            yt=y[i]+tt*log(ran1(&idum));        More thermal fluctuations.
            if (yt <= ylo) {
                ilo=i;
                ylo=yt;
            }
            if (yt > yhi) {
                ynhi=yhi;
                ihi=i;
                yhi=yt;
            } else if (yt > ynhi) {
                ynhi=yt;
            }
        }
        rtol=2.0*fabs(yhi-ylo)/(fabs(yhi)+fabs(ylo));
        Compute the fractional range from highest to lowest and return if satisfactory.
        if (rtol < ftol || *iter < 0) {         If returning, put best point and value in
            swap=y[1];                              slot 1.
            y[1]=y[ilo];
            y[ilo]=swap;
            for (n=1;n<=ndim;n++) {
                swap=p[1][n];
                p[1][n]=p[ilo][n];
                p[ilo][n]=swap;
            }
            break;
        }
        *iter -= 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=amotsa(p,y,psum,ndim,pb,yb,funk,ihi,&yhi,-1.0);
        if (ytry <= ylo) {
            Gives a result better than the best point, so try an additional extrapolation by a
            factor of 2.
            ytry=amotsa(p,y,psum,ndim,pb,yb,funk,ihi,&yhi,2.0);
        } else if (ytry >= ynhi) {
            The reflected point is worse than the second-highest, so look for an intermediate
```

```
            lower point, i.e., do a one-dimensional contraction.
            ysave=yhi;
            ytry=amotsa(p,y,psum,ndim,pb,yb,funk,ihi,&yhi,0.5);
            if (ytry >= ysave) {                Can't seem to get rid of that high point.
                for (i=1;i<=mpts;i++) {             Better contract around the lowest
                    if (i != ilo) {                 (best) point.
                        for (j=1;j<=ndim;j++) {
                            psum[j]=0.5*(p[i][j]+p[ilo][j]);
                            p[i][j]=psum[j];
                        }
                        y[i]=(*funk)(psum);
                    }
                }
                *iter -= ndim;
                GET_PSUM                        Recompute psum.
            }
        } else ++(*iter);                       Correct the evaluation count.
    }
    free_vector(psum,1,ndim);
}
```

```
#include <math.h>
#include "nrutil.h"

extern long idum;                       Defined and initialized in main.
extern float tt;                        Defined in amebsa.

float amotsa(float **p, float y[], float psum[], int ndim, float pb[],
    float *yb, float (*funk)(float []), int ihi, float *yhi, 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.
{
    float ran1(long *idum);
    int j;
    float fac1,fac2,yflu,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);
    if (ytry <= *yb) {                  Save the best-ever.
        for (j=1;j<=ndim;j++) pb[j]=ptry[j];
        *yb=ytry;
    }
    yflu=ytry-tt*log(ran1(&idum));      We added a thermal fluctuation to all the current
    if (yflu < *yhi) {                      vertices, but we subtract it here, so as to give
        y[ihi]=ytry;                        the simplex a thermal Brownian motion: It
        *yhi=yflu;                          likes to accept any suggested change.
        for (j=1;j<=ndim;j++) {
            psum[j] += ptry[j]-p[ihi][j];
            p[ihi][j]=ptry[j];
        }
    }
    free_vector(ptry,1,ndim);
    return yflu;
}
```

There is not yet enough practical experience with the method of simulated annealing to say definitively what its future place among optimization methods

will be. The method has several extremely attractive features, rather unique when compared with other optimization techniques.

First, it is not "greedy," in the sense that it is not easily fooled by the quick payoff achieved by falling into unfavorable local minima. Provided that sufficiently general reconfigurations are given, it wanders freely among local minima of depth less than about T. As T is lowered, the number of such minima qualifying for frequent visits is gradually reduced.

Second, configuration decisions tend to proceed in a logical order. Changes that cause the greatest energy differences are sifted over when the control parameter T is large. These decisions become more permanent as T is lowered, and attention then shifts more to smaller refinements in the solution. For example, in the traveling salesman problem with the Mississippi River twist, if λ is large, a decision to cross the Mississippi only twice is made at high T, while the specific routes on each side of the river are determined only at later stages.

The analogies to thermodynamics may be pursued to a greater extent than we have done here. Quantities analogous to specific heat and entropy may be defined, and these can be useful in monitoring the progress of the algorithm towards an acceptable solution. Information on this subject is found in [1].

CITED REFERENCES AND FURTHER READING:

Kirkpatrick, S., Gelatt, C.D., and Vecchi, M.P. 1983, *Science*, vol. 220, pp. 671–680. [1]

Kirkpatrick, S. 1984, *Journal of Statistical Physics*, vol. 34, pp. 975–986. [2]

Vecchi, M.P. and Kirkpatrick, S. 1983, *IEEE Transactions on Computer Aided Design*, vol. CAD-2, pp. 215–222. [3]

Otten, R.H.J.M., and van Ginneken, L.P.P.P. 1989, *The Annealing Algorithm* (Boston: Kluwer) [contains many references to the literature]. [4]

Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller A., and Teller, E. 1953, *Journal of Chemical Physics*, vol. 21, pp. 1087–1092. [5]

Lin, S. 1965, *Bell System Technical Journal*, vol. 44, pp. 2245–2269. [6]

Vanderbilt, D., and Louie, S.G. 1984, *Journal of Computational Physics*, vol. 56, pp. 259–271. [7]

Bohachevsky, I.O., Johnson, M.E., and Stein, M.L. 1986, *Technometrics*, vol. 28, pp. 209–217. [8]

Corana, A., Marchesi, M., Martini, C., and Ridella, S. 1987, *ACM Transactions on Mathematical Software*, vol. 13, pp. 262–280. [9]

Bélisle, C.J.P., Romeijn, H.E., and Smith, R.L. 1990, Technical Report 90–25, Department of Industrial and Operations Engineering, University of Michigan, submitted to *Mathematical Programming*. [10]

Christofides, N., Mingozzi, A., Toth, P., and Sandi, C. (eds.) 1979, *Combinatorial Optimization* (London and New York: Wiley-Interscience) [not simulated annealing, but other topics and algorithms].