Chapter 10. Minimization or Maximization of Functions

10.0 Introduction

In a nutshell: You are given a single function f which 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 person'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.) Virtually nothing is known about finding global extrema in general. There are two standard heuristics that everyone uses: (i) find local extrema starting from widely varying starting values of the independent variables, 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. There are tantalizing hints that so-called "annealing methods" may lead to important progress on the global extremization problem (see below).

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 non-negative 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

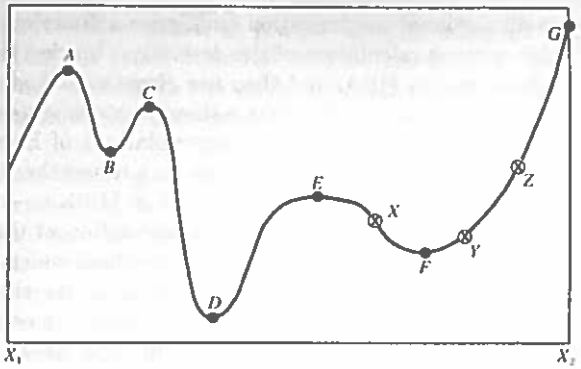

Figure 10.0.1. Extrema of a function in an interval. Points A, C, and E are local, but not global maxima. Points B and F are local, but not global minima. The global maximum occurs at G, which is on the boundary of the interval so that the derivative of the function need not vanish there. The global 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.

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.

One other section, §10.9, also lies outside of our main thrust, but for exactly the opposite reason: so-called "annealing methods" are so new that we do not yet know where they will ultimately fit into the scheme of things. We do know that they have solved some problems previously thought to be practically insoluble, and that they have a direct bearing on the problem of finding global extrema in the presence of large numbers of undesired local extrema. We have therefore chosen to include some discussion of this new area.

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 which 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 somewhat neglected *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.

REFERENCES AND FURTHER READING:

Acton, Forman S. 1970, *Numerical Methods That Work* (New York: Harper and Row), Chapter 17.

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

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

Dahlquist, Germund, and Bjorck, Ake. 1974, *Numerical Methods* (Englewood Cliffs, N.J.: Prentice-Hall), Chapter 10.

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

10.1 Golden Section Search in One Dimension

Recall how the bisection method finds roots of functions in one dimension (§9.2): 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, $a < b$, 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, $a < b < c$, 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 $a < b < x$; 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} (single precision) or 10^{-15} (double precision). 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×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 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.

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

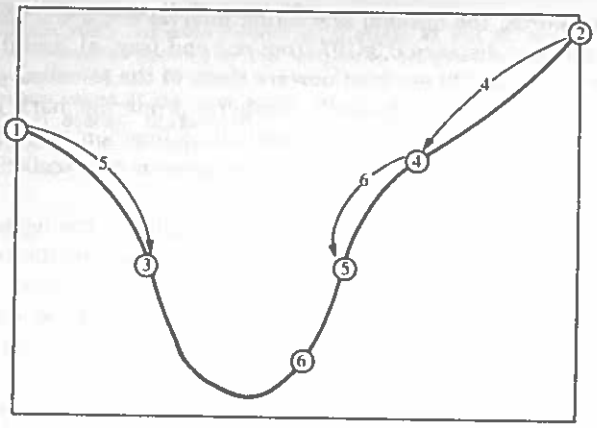

Figure 10.1.1. Successive bracketing of a minimum. The minimum is originally bracketed by points 1,3,2. The function is evaluated at 4, which replaces 2; then at 5, which replaces 1; then at 6, which replaces 4. The rule at each stage is to keep a center point that is lower than the two outside points.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

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 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) yield 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 which 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 non-bracketing 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 step size 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:

```
      SUBROUTINE MNBRAK(AX,BX,CX,FA,FB,FC,FUNC)
        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 which bracket a minimum of the function.  Also returned
        are the function values at the three points, FA, FB, and FC.
      PARAMETER (GOLD=1.618034, GLIMIT=100., TINY=1.E-20)
        The first parameter is the default ratio by which successive intervals are magnified; the
        second is the maximum magnification allowed for a parabolic-fit step.
      FA=FUNC(AX)
      FB=FUNC(BX)
      IF(FB.GT.FA)THEN                  Switch roles of A and B so that we can go downhill in the direction
        DUM=AX                                from A to B.
        AX=BX
        BX=DUM
        DUM=FB
        FB=FA
        FA=DUM
      ENDIF
      CX=BX+GOLD*(BX-AX)                First guess for C.
      FC=FUNC(CX)
    1 IF(FB.GE.FC)THEN                  "DO WHILE": keep returning here until we bracket.
        R=(BX-AX)*(FB-FC)               Compute U by parabolic extrapolation from A,B,C.  TINY is used to
        Q=(BX-CX)*(FB-FA)                     prevent any possible division by zero.
        U=BX-((BX-CX)*Q-(BX-AX)*R)/(2.*SIGN(MAX(ABS(Q-R),TINY),Q-R))
        ULIM=BX+GLIMIT*(CX-BX)          We won't go farther than this.  Now to test various possibilities:
        IF((BX-U)*(U-CX).GT.0.)THEN           Parabolic U is between B and C: try it.
          FU=FUNC(U)
          IF(FU.LT.FC)THEN              Got a minimum between B and C.
            AX=BX
            FA=FB
            BX=U
            FB=FU
            GO TO 1                     (which will exit).
          ELSE IF(FU.GT.FB)THEN         Got a minimum between between A and U.
            CX=U
            FC=FU
            GO TO 1                     (which will exit).
          ENDIF
          U=CX+GOLD*(CX-BX)             Parabolic fit was no use.  Use default magnification.
          FU=FUNC(U)
        ELSE IF((CX-U)*(U-ULIM).GT.0.)THEN          Parabolic fit is between C and its allowed limit.
          FU=FUNC(U)
          IF(FU.LT.FC)THEN
            BX=CX
            CX=U
            U=CX+GOLD*(CX-BX)
            FB=FC
            FC=FU
            FU=FUNC(U)
          ENDIF
        ELSE IF((U-ULIM)*(ULIM-CX).GE.0.)THEN       Limit parabolic U to maximum allowed value.
          U=ULIM
          FU=FUNC(U)
        ELSE                            Reject parabolic U, use default magnification.
          U=CX+GOLD*(CX-BX)
          FU=FUNC(U)
        ENDIF
        AX=BX                           Eliminate oldest point and continue.
        BX=CX
        CX=U
        FA=FB
        FB=FC
        FC=FU
        GO TO 1
```

```
      ENDIF
      RETURN
      END
```

(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, be advised. The underlying ideas are quite simple.)

Routine for Golden Section Search

```
      FUNCTION GOLDEN(AX,BX,CX,F,TOL,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.
      PARAMETER (R=.61803399,C=1.-R)  Golden ratios.
      X0=AX                           At any given time we will keep track of four points, X0,X1,X2,X3.
      X3=CX
      IF(ABS(CX-BX).GT.ABS(BX-AX))THEN    Make X0 to X1 the smaller segment,
        X1=BX
        X2=BX+C*(CX-BX)                   and fill in the new point to be tried.
      ELSE
        X2=BX
        X1=BX-C*(BX-AX)
      ENDIF
      F1=F(X1)
      F2=F(X2)                        The initial function evaluations. Note that we never need to evaluate
1     IF(ABS(X3-X0).GT.TOL*(ABS(X1)+ABS(X2)))THEN    the function at the original endpoints.
        IF(F2.LT.F1)THEN                  Do-while loop: we keep returning here.
          X0=X1                           One possible outcome,
          X1=X2                           its housekeeping,
          X2=R*X1+C*X3
          F0=F1
          F1=F2
          F2=F(X2)                        and a new function evaluation.
        ELSE                              The other outcome,
          X3=X2
          X2=X1
          X1=R*X2+C*X0
          F3=F2
          F2=F1
          F1=F(X1)                        and its new function evaluation.
        ENDIF
      GOTO 1                          Back to see if we are done.
      ENDIF
      IF(F1.LT.F2)THEN                We are done. Output the best of the two current values.
        GOLDEN=F1
        XMIN=X1
      ELSE
        GOLDEN=F2
        XMIN=X2
      ENDIF
      RETURN
      END
```

10.2 Parabolic Interpolation and Brent's Method in One-Dimension

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

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

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

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

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

Brent's method (Brent, 1973) is up to the task in all particulars. At any particular stage, it is keeping track of six function points (not necessarily all distinct), a, b, u, v, w and x, defined as follows: the minimum is bracketed between a and b; x is the point with the very least function value found so far (or the most recent one in case of a tie); w is the point with the second least function value; v is the previous value of w; u is the point at which the function was evaluated most recently. Also appearing in the algorithm is the point x_m, the midpoint between a and b; however the function is not evaluated there.

You can read the code below to understand the method's logical organization. Mention of a few general principles here may, however, be helpful: Parabolic interpolation is attempted, fitting through the points x, v, and w. To be acceptable, the parabolic step must (i) fall within the bounding interval (a,b), and (ii) imply a movement from the best current value x that is *less*

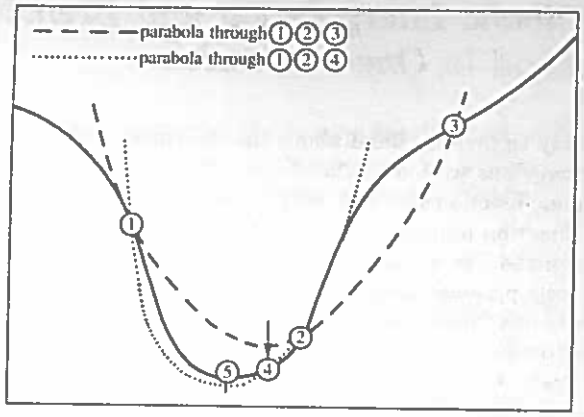

Figure 10.2.1. Convergence to a minimum by inverse parabolic interpolation. A parabola (dashed line) is drawn through the three original points 1,2,3 on the given function (solid line). The function is evaluated at the parabola's minimum, 4, which replaces point 3. A new parabola (dotted line) is drawn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

than half the movement of the *step before last*. This second criterion insures that the parabolic steps are actually converging to something, rather than, say, bouncing around in some nonconvergent limit cycle. In the worst possible case, where the parabolic steps are acceptable but useless, the method will approximately alternate between parabolic steps and golden sections, converging in due course by virtue of the latter. The reason for comparing to the step *before* last seems essentially heuristic: experience shows that it is better not to "punish" the algorithm for a single bad step if it can make it up on the next one.

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

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

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

```
FUNCTION BRENT(AX,BX,CX,F,TOL,XMIN)
    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.
PARAMETER (ITMAX=100,CGOLD=.3819660,ZEPS=1.0E-10)
```



```
    Maximum allowed number of iterations, golden ratio, and a small number which protects
    against trying to achieve fractional accuracy for a minimum that happens to be exactly
    zero.
A=MIN(AX,CX)                    A and B must be in ascending order, though the input abscissas need
B=MAX(AX,CX)                        not be.
V=BX                            Initializations...
W=V
X=V
E=0.                            This will be the distance moved on the step before last.
FX=F(X)
FV=FX
FW=FX
DO 11 ITER=1,ITMAX              Main program loop.
    XM=0.5*(A+B)
    TOL1=TOL*ABS(X)+ZEPS
    TOL2=2.*TOL1
    IF(ABS(X-XM).LE.(TOL2-.5*(B-A))) GOTO 3 Test for done here.
    IF(ABS(E).GT.TOL1) THEN         Construct a trial parabolic fit.
        R=(X-W)*(FX-FV)
        Q=(X-V)*(FX-FW)
        P=(X-V)*Q-(X-W)*R
        Q=2.*(Q-R)
        IF(Q.GT.0.)  P=-P
        Q=ABS(Q)
        ETEMP=E
        E=D
*       IF(ABS(P).GE.ABS(.5*Q*ETEMP).OR.P.LE.Q*(A-X).OR.
*           P.GE.Q*(B-X)) GOTO 1
    The above conditions determine the acceptability of the parabolic fit. Here it is o.k.:
        D=P/Q                       Take the parabolic step.
        U=X+D
        IF(U-A.LT.TOL2 .OR. B-U.LT.TOL2) D=SIGN(TOL1,XM-X)
        GOTO 2                      Skip over the golden section step.
    ENDIF
1   IF(X.GE.XM) THEN                We arrive here for a golden section step, which we take into the larger
        E=A-X                           of the two segments.
    ELSE
        E=B-X
    ENDIF
    D=CGOLD*E                       Take the golden section step.
2   IF(ABS(D).GE.TOL1) THEN         Arrive here with D computed either from parabolic fit, or else from
        U=X+D                           golden section.
    ELSE
        U=X+SIGN(TOL1,D)
    ENDIF
    FU=F(U)                         This is the one function evaluation per iteration,
    IF(FU.LE.FX) THEN               and now we have to decide what to do with our function evaluation.
        IF(U.GE.X) THEN                 Housekeeping follows:
            A=X
        ELSE
            B=X
        ENDIF
        V=W
        FV=FW
        W=X
        FW=FX
        X=U
        FX=FU
    ELSE
        IF(U.LT.X) THEN
            A=U
        ELSE
            B=U
        ENDIF
```

```
        IF(FU.LE.FW .OR. W.EQ.X) THEN
          V=W
          FV=FW
          W=U
          FW=FU
        ELSE IF(FU.LE.FV .OR. V.EQ.X .OR. V.EQ.W) THEN
          V=U
          FV=FU
        ENDIF
      ENDIF                        Done with housekeeping. Back for another iteration.
    11 CONTINUE
    PAUSE 'Brent exceed maximum iterations.'
3   XMIN=X                         Arrive here ready to exit with best values.
    BRENT=FX
    RETURN
    END
```

REFERENCES AND FURTHER READING:

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

Forsythe, George E., Malcolm, Michael A., and Moler, Cleve B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, N.J.: Prentice-Hall), §8.2.

10.3 One-Dimensional Search with First Derivatives

Here we want to accomplish precisely the same goal as in the previous section, namely to isolate a functional minimum that is bracketed by the triplet of abscissas $a < b < c$, but utilizing an additional capability to compute the function's first derivative as well as its value.

In principle, we might simply search for a zero of the derivative, ignoring the function value information, using a root finder like RTFLSP or ZBRENT (§9.2). It doesn't take long to reject *that* idea: How do we distinguish maxima from minima? Where do we go from initial conditions where the derivatives on one or both of the outer bracketing points indicate that "downhill" is in the direction *out* of the bracketed interval?

We don't want to give up our strategy of maintaining a rigorous bracket on the minimum at all times. The only way to keep such a bracket is to update it using function (not derivative) information, with the central point in the bracketing triplet always that with the lowest function value. Therefore the role of the derivatives can only be to help us choose new trial points within the bracket.

One school of thought is to "use everything you've got": Compute a polynomial of relatively high order (cubic or above) which agrees with some number of previous function and derivative evaluations. For example, there is a unique cubic that agrees with function and derivative at two points,

and one can jump to the interpolated minimum of that cubic (if there is a minimum within the bracket). Suggested by Davidon and others, formulas for this tactic are given in Acton.

We like to be more conservative than this. Once superlinear convergence sets in, it hardly matters whether its order is moderately lower or higher. In practical problems that we have met, most function evaluations are spent in getting globally close enough to the minimum for superlinear convergence to commence. So we are more worried about all the funny "stiff" things that high order polynomials can do (cf. Figure 3.0.1b), and about their sensitivities to roundoff error.

This leads us to use derivative information only as follows: The sign of the derivative at the central point of the bracketing triplet $a < b < c$ indicates uniquely whether the next test point should be taken in the interval (a, b) or in the interval (b, c). The value of this derivative and of the derivative at the second-best-so-far point are extrapolated to zero by the secant method (inverse linear interpolation), which by itself is superlinear of order 1.618. (The golden mean again: see Acton, p. 57.) We impose the same sort of restrictions on this new trial point as in Brent's method. If the trial point must be rejected, we *bisect* the interval under scrutiny.

Yes, we are fuddy-duddies when it comes to making flamboyant use of derivative information in one-dimensional minimization. But we have had a bellyful of functions whose computed "derivatives" *don't* integrate up to the function value and *don't* accurately point the way to the minimum, usually because of roundoff errors, sometimes because of truncation error in the method of derivative evaluation.

You will see that the following routine is closely modeled on BRENT in the previous section.

```
      FUNCTION DBRENT(AX,BX,CX,F,DF,TOL,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.
      PARAMETER (ITMAX=100,ZEPS=1.0E-10)
        Comments following will point out only differences from the routine BRENT. Read that
        routine first.
      LOGICAL OK1,OK2                  Will be used as flags for whether proposed steps are acceptable or
      A=MIN(AX,CX)                       not
      B=MAX(AX,CX)
      V=BX
      W=V
      X=V
      E=0.
      FX=F(X)
      FV=FX
      FW=FX
      DX=DF(X)                         All our housekeeping chores are doubled by the necessity of moving
      DV=DX                              derivative values around as well as function values.
      DW=DX
      DO 11 ITER=1,ITMAX
        XM=0.5*(A+B)
```

```
      TOL1=TOL*ABS(X)+ZEPS
      TOL2=2.*TOL1
      IF(ABS(X-XM).LE.(TOL2-.5*(B-A))) GOTO 3
      IF(ABS(E).GT.TOL1) THEN
        D1=2.*(B-A)           Initialize these D's to an out-of-bracket value.
        D2=D1
        IF(DW.NE.DX) D1=(W-X)*DX/(DX-DW)      Secant method.
        IF(DV.NE.DX) D2=(V-X)*DX/(DX-DV)      Secant method with the other stored point.
```

Which of these two estimates of D shall we take? We will insist that they be within the bracket, and on the side pointed to by the derivative at X:

```
        U1=X+D1
        U2=X+D2
        OK1=((A-U1)*(U1-B).GT.0.).AND.(DX*D1.LE.0.)
        OK2=((A-U2)*(U2-B).GT.0.).AND.(DX*D2.LE.0.)
        OLDE=E                Movement on the step before last.
        E=D
        IF(.NOT.(OK1.OR.OK2))THEN     Take only an acceptable D, and if both are acceptable, then
          GO TO 1                     take the smallest one.
        ELSE IF (OK1.AND.OK2)THEN
          IF(ABS(D1).LT.ABS(D2))THEN
            D=D1
          ELSE
            D=D2
          ENDIF
        ELSE IF (OK1)THEN
          D=D1
        ELSE
          D=D2
        ENDIF
        IF(ABS(D).GT.ABS(0.5*OLDE))GO TO 1
        U=X+D
        IF(U-A.LT.TOL2 .OR. B-U.LT.TOL2) D=SIGN(TOL1,XM-X)
        GOTO 2
      ENDIF
1     IF(DX.GE.0.)  THEN          Decide which segment by the sign of the derivative.
        E=A-X
      ELSE
        E=B-X
      ENDIF
      D=0.5*E                     Bisect, not golden section.
2     IF(ABS(D).GE.TOL1) THEN
        U=X+D
        FU=F(U)
      ELSE
        U=X+SIGN(TOL1,D)
        FU=F(U)
        IF(FU.GT.FX)GO TO 3       If the minimum step in the downhill direction takes us uphill, then we
      ENDIF                             are done.
      DU=DF(U)                    Now all the housekeeping, sigh.
      IF(FU.LE.FX) THEN
        IF(U.GE.X) THEN
          A=X
        ELSE
          B=X
        ENDIF
        V=W
        FV=FW
        DV=DW
        W=X
        FW=FX
        DW=DX
        X=U
        FX=FU
        DX=DU
      ELSE
        IF(U.LT.X) THEN
          A=U
        ELSE
          B=U
        ENDIF
        IF(FU.LE.FW .OR. W.EQ.X) THEN
          V=W
          FV=FW
          DV=DW
          W=U
          FW=FU
          DW=DU
        ELSE IF(FU.LE.FV .OR. V.EQ.X .OR. V.EQ.W) THEN
          V=U
          FV=FU
          DV=DU
        ENDIF
      ENDIF
    11CONTINUE
    PAUSE 'DBRENT exceeded maximum iterations.'
3   XMIN=X
    DBRENT=FX
    RETURN
    END
```


REFERENCES AND FURTHER READING:

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

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

10.4 Downhill Simplex Method in Multidimensions

With this section we begin consideration of multidimensional minimization, that is, finding the minimum of a function of more than one independent variable. This section stands apart from those which follow, however: All of the algorithms after this section will make explicit use of the one-dimensional minimization algorithms of §10.1, §10.2, or §10.3 as a part of their computational strategy. This section implements an entirely self-contained strategy, in which one-dimensional minimization does not figure.

The *downhill simplex method* is due to Nelder and Mead (1965). The method requires only function evaluations, not derivatives. It is not very efficient in terms of the number of function evaluations that it requires. Powell's method (§10.5) is almost surely faster in all likely applications. However the downhill simplex method may frequently be the *best* method to use if the figure of merit is "get something working quickly" for a problem whose computational burden is small.

The method has a geometrical naturalness about it which makes it delightful to describe or work through:

A *simplex* is the geometrical figure consisting, in N dimensions, of $N+1$ points (or vertices) and all their interconnecting line segments, polygonal faces, etc. In two dimensions, a simplex is a triangle. In three dimensions it is a tetrahedron, not necessarily the regular tetrahedron. (The *simplex method* of linear programming also makes use of the geometrical concept of a simplex. Otherwise it is completely unrelated to the algorithm that we are describing in this section.) In general we are only interested in simplexes that are nondegenerate, i.e. which enclose a finite inner N-dimensional volume. If any point of a nondegenerate simplex is taken as the origin, then the N other points define vector directions that span the N-dimensional vector space.

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

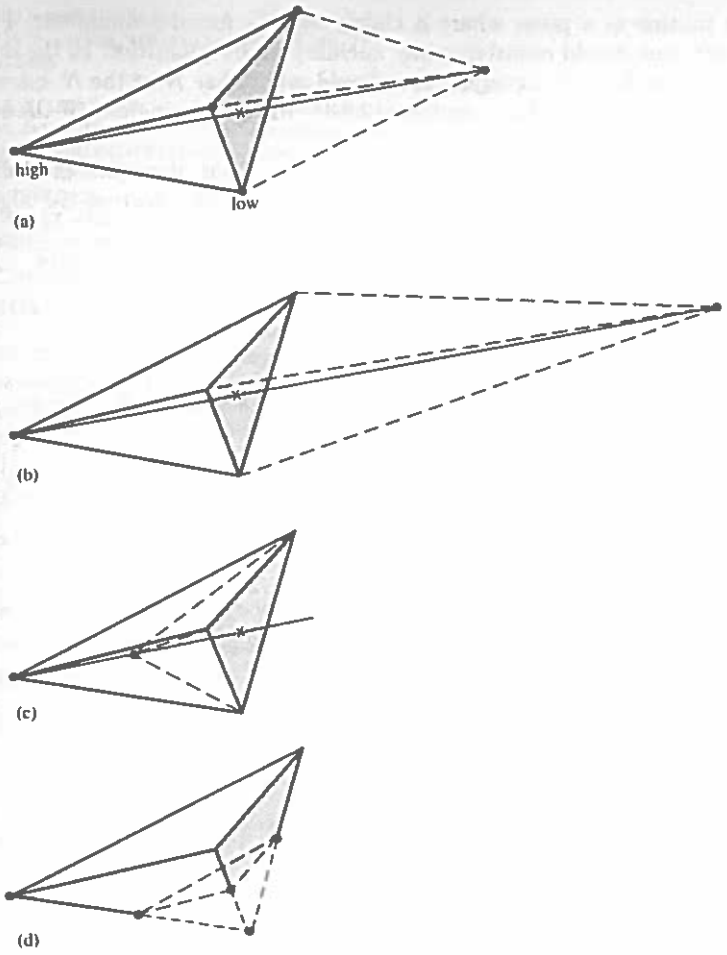

Figure 10.4.1. Possible outcomes for a step in the downhill simplex method. The simplex at the beginning of the step, here a tetrahedron, is drawn with solid lines. The simplex at the end of the step (drawn dashed) can be either (a) a reflection away from the high point, (b) a reflection and expansion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

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 minimiza-

tion 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:

```
      SUBROUTINE AMOEBA(P,Y,MP,NP,NDIM,FTOL,FUNK,ITER)
        Multidimensional minimization of the function FUNK(X) where X is an NDIM-dimensional
        vector, by the downhill simplex method of Nelder and Mead. Input is a matrix P whose
        NDIM+1 rows are NDIM-dimensional vectors which are the vertices of the starting simplex.
        [Logical dimensions of P are P(NDIM+1,NDIM), physical dimensions are input as P(MP,NP)]
        Also input is the vector Y of length 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 ITER gives the number of iterations taken.
      PARAMETER (NMAX=20,ALPHA=1.0,BETA=0.5,GAMMA=2.0,ITMAX=500)
        Expected maximum number of dimensions, three parameters which define the expansions
        and contractions, and maximum allowed number of iterations.
      DIMENSION P(MP,NP),Y(MP),PR(NMAX),PRR(NMAX),PBAR(NMAX)
      MPTS=NDIM+1                          Note that MP is the physical dimension corresponding to the logical
      ITER=0                                     dimension MPTS, NP to NDIM.
    1 ILO=1                                First we must determine which point is the highest (worst), next-
      IF(Y(1).GT.Y(2))THEN                       highest, and lowest (best).
        IHI=1
        INHI=2
      ELSE
        IHI=2
        INHI=1
      ENDIF
      DO 11 I=1,MPTS                       by looping over the points in the simplex.
        IF(Y(I).LT.Y(ILO)) ILO=I
        IF(Y(I).GT.Y(IHI))THEN
          INHI=IHI
          IHI=I
        ELSE IF(Y(I).GT.Y(INHI))THEN
          IF(I.NE.IHI) INHI=I
        ENDIF
      11 CONTINUE
        Compute the fractional range from highest to lowest and return if satisfactory.
      RTOL=2.*ABS(Y(IHI)-Y(ILO))/(ABS(Y(IHI))+ABS(Y(ILO)))
      IF(RTOL.LT.FTOL)RETURN
      IF(ITER.EQ.ITMAX) PAUSE 'Amoeba exceeding maximum iterations.'
      ITER=ITER+1
      DO 12 J=1,NDIM
        PBAR(J)=0.
      12 CONTINUE                          Begin a new iteration. Compute the vector average of all points except
      DO 14 I=1,MPTS                             the highest, i.e. the center of the "face" of the simplex across
        IF(I.NE.IHI)THEN                         from the high point. We will subsequently explore along the
          DO 13 J=1,NDIM                         ray from the high point through that center.
            PBAR(J)=PBAR(J)+P(I,J)
          13 CONTINUE
        ENDIF
      14 CONTINUE
      DO 15 J=1,NDIM                       Extrapolate by a factor ALPHA through the face, i.e. reflect the simplex
        PBAR(J)=PBAR(J)/NDIM                     from the high point.
        PR(J)=(1.+ALPHA)*PBAR(J)-ALPHA*P(IHI,J)
      15 CONTINUE
      YPR=FUNK(PR)                         Evaluate the function at the reflected point.
      IF(YPR.LE.Y(ILO))THEN                Gives a result better than the best point, so
        DO 16 J=1,NDIM                     try an additional extrapolation by a factor GAMMA.
          PRR(J)=GAMMA*PR(J)+(1.-GAMMA)*PBAR(J)
        16 CONTINUE
        YPRR=FUNK(PRR)                     and check out the function there.
        IF(YPRR.LT.Y(ILO))THEN             The additional extrapolation succeeded,
          DO 17 J=1,NDIM                   and replaces the high point.
            P(IHI,J)=PRR(J)
          17 CONTINUE
          Y(IHI)=YPRR
        ELSE                               The additional extrapolation failed,
          DO 18 J=1,NDIM                   but we can still use the reflected point.
            P(IHI,J)=PR(J)
          18 CONTINUE
          Y(IHI)=YPR
        ENDIF
      ELSE IF(YPR.GE.Y(INHI))THEN          The reflected point is worse than the second-highest.
        IF(YPR.LT.Y(IHI))THEN              If it's better than the highest, then replace the highest.
          DO 19 J=1,NDIM
            P(IHI,J)=PR(J)
          19 CONTINUE
          Y(IHI)=YPR
        ENDIF
        DO 21 J=1,NDIM                     but look for an intermediate lower point,
          PRR(J)=BETA*P(IHI,J)+(1.-BETA)*PBAR(J)
        21 CONTINUE                        in other words, perform a contraction of the simplex along one di-
        YPRR=FUNK(PRR)                           mension. Then evaluate the function.
        IF(YPRR.LT.Y(IHI))THEN             Contraction gives an improvement,
          DO 22 J=1,NDIM                   so accept it.
            P(IHI,J)=PRR(J)
          22 CONTINUE
          Y(IHI)=YPRR
        ELSE                               Can't seem to get rid of that high point. Better contract around the
          DO 24 I=1,MPTS                           lowest (best) point.
            IF(I.NE.ILO)THEN
              DO 23 J=1,NDIM
                PR(J)=0.5*(P(I,J)+P(ILO,J))
                P(I,J)=PR(J)
              23 CONTINUE
              Y(I)=FUNK(PR)
            ENDIF
          24 CONTINUE
        ENDIF
      ELSE                                 We arrive here if the original reflection gives a middling point. Replace
        DO 25 J=1,NDIM                           the old high point and continue
          P(IHI,J)=PR(J)
        25 CONTINUE
        Y(IHI)=YPR
      ENDIF
      GO TO 1                              for the test of doneness and the next iteration.
      END
```

REFERENCES AND FURTHER READING:

Nelder, J.A., and Mead, R. 1965, *Computer Journal*, vol. 7, p. 308.

Yarbro, L.A., and Deming, S.N. 1974, *Analytica Chim. Acta*, vol. 73, p. 391.

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

10.5 Direction Set (Powell's) Methods in Multidimensions

We know (§10.1–§10.3) how to minimize a function of one variable. If we start at a point $\mathbf{P}$ in N-dimensional space, and proceed from there in some vector direction $\mathbf{n}$, then any function of N variables $f(\mathbf{P})$ can be minimized along the line $\mathbf{n}$ by our one-dimensional methods. One can dream up various multidimensional minimization methods which consist of sequences of such line minimizations. Different methods will differ only by how, at each stage, they choose the next direction $\mathbf{n}$ to try. All such methods presume the existence of a "black-box" subalgorithm, 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 $\mathbf{P}$ and $\mathbf{n}$, and the function f, find the scalar λ that minimizes $f(\mathbf{P}+\lambda\mathbf{n})$. Replace $\mathbf{P}$ by $\mathbf{P}+\lambda\mathbf{n}$. Replace $\mathbf{n}$ 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. 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 dumb 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; by Murphy's Law, you should count on it.

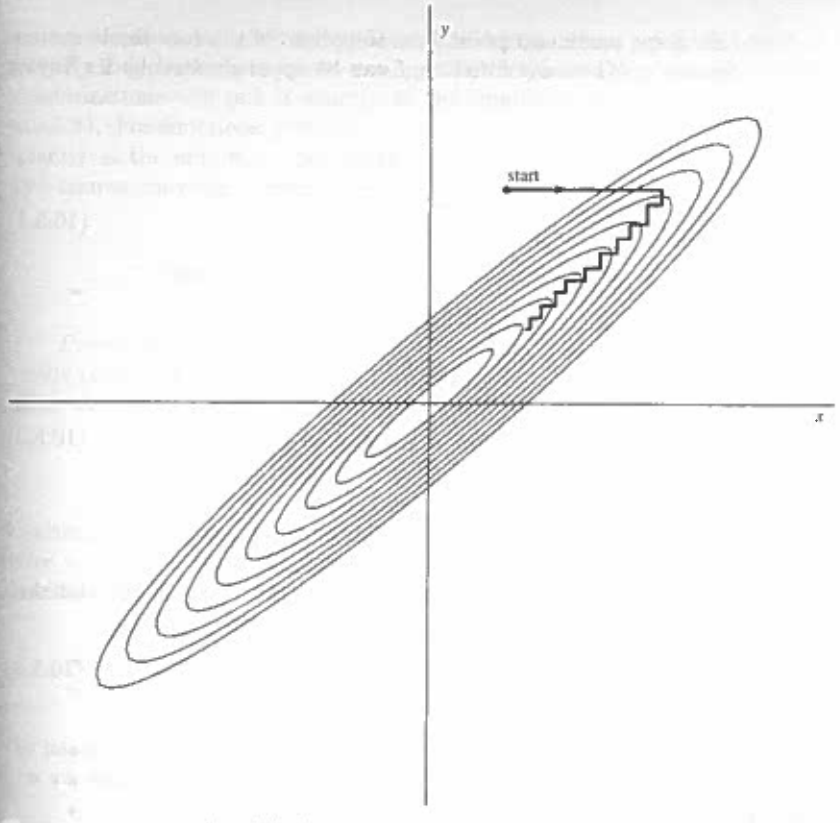

Figure 10.5.1. Successive minimizations along coordinate directions in a long, narrow "valley" (shown as contour lines). Unless the valley is optimally oriented, this method is extremely inefficient, taking many tiny steps to get to the minimum, crossing and re-crossing the principal axis.

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.

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

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

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 which 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 which 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 (1973) 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 A (which he gives a procedure for determining). The calculation is essentially a singular value decomposition algorithm (see §2.9). 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 his book 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 which we now implement. (It is also the version of Powell's method given in Acton, from which parts of the following discussion are drawn.)

Powell's Method 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 **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.

```
      SUBROUTINE POWELL(P,XI,N,NP,FTOL,ITER,FRET)
        Minimization of a function FUNC of N variables. (FUNC is not an argument, it is a fixed
        function name.) Input consists of an initial starting point P that is a vector of length N,
        an initial matrix XI whose logical dimensions are N by N, physical dimensions NP by NP,
        and 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.
      PARAMETER (NMAX=20,ITMAX=200)     Maximum expected value of N, and maximum allowed iterations.
      DIMENSION P(NP),XI(NP,NP),PT(NMAX),PTT(NMAX),XIT(NMAX)
      FRET=FUNC(P)
      DO 11 J=1,N                       Save the initial point.
        PT(J)=P(J)
   11 CONTINUE
      ITER=0
    1 ITER=ITER+1
      FP=FRET
      IBIG=0
      DEL=0.                            Will be the biggest function decrease.
      DO 13 I=1,N                       In each iteration, loop over all directions in the set.
        DO 12 J=1,N                     Copy the direction,
          XIT(J)=XI(J,I)
   12   CONTINUE
        FPTT=FRET
        CALL LINMIN(P,XIT,N,FRET)       minimize along it,
        IF(ABS(FPTT-FRET).GT.DEL)THEN   and record it if it is the largest decrease so far.
          DEL=ABS(FPTT-FRET)
          IBIG=I
        ENDIF
   13 CONTINUE
      IF(2.*ABS(FP-FRET).LE.FTOL*(ABS(FP)+ABS(FRET)))RETURN   Termination criterion.
      IF(ITER.EQ.ITMAX) PAUSE 'Powell exceeding maximum iterations.'
      DO 14 J=1,N                       Construct the extrapolated point and the average direction moved.
        PTT(J)=2.*P(J)-PT(J)                Save the old starting point.
        XIT(J)=P(J)-PT(J)
        PT(J)=P(J)
   14 CONTINUE
      FPTT=FUNC(PTT)                    Function value at extrapolated point.
      IF(FPTT.GE.FP)GO TO 1             One reason not to use new direction.
      T=2.*(FP-2.*FRET+FPTT)*(FP-FRET-DEL)**2-DEL*(FP-FPTT)**2
      IF(T.GE.0.)GO TO 1                Other reason not to use new direction.
      CALL LINMIN(P,XIT,N,FRET)         Move to the minimum of the new direction,
      DO 15 J=1,N                       and save the new direction.
```

```
      XI(J,IBIG)=XIT(J)
   15 CONTINUE
GO TO 1                    Back for another iteration.
END
```

Implementation of Line Minimization

In the above routine, you might have wondered why we didn't make the function name FUNC an argument of the routine. The reason is buried in a slightly dirty FORTRAN practicality in our implementation of LINMIN.

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 "DO K=1,N" 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 FUNC along the line going through the point P in the direction XI. LINMIN communicates with F1DIM through a common block. (Woe betide the Pascal programmer!) It then calls our familiar one-dimensional routines MNBRAK (§10.1) and BRENT (§10.2) and instructs them to minimize F1DIM.

Still following? Then try this: BRENT is passed the function name F1DIM, which it dutifully calls. But there is no way to signal to F1DIM that it is supposed to use your function name, which could have been passed to LINMIN as an argument. Therefore, we have to make F1DIM use a *fixed* function name, namely FUNC. The situation is reminiscent of Henry Ford's black automobile: POWELL will minimize any function, as long as it is named FUNC. Needed to remedy this situation is a way to pass a function name through a common block; this is lacking in FORTRAN. And Pascal is even worse; the Pascal programmer probably has no idea what we are talking about!

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 from hither to yon and back again. That should not normally be a significant addition to the overall computational burden, but we cannot disguise its inelegance.

```
SUBROUTINE LINMIN(P,XI,N,FRET)
    Given an N dimensional point P and an N dimensional direction XI, 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.
PARAMETER (NMAX=50,TOL=1.E-4)
    Maximum anticipated N, and TOL passed to BRENT.
EXTERNAL F1DIM
DIMENSION P(N),XI(N)
COMMON /F1COM/ NCOM,PCOM(NMAX),XICOM(NMAX)
NCOM=N                     Set up the common block.
DO 11 J=1,N
```



```
      PCOM(J)=P(J)
      XICOM(J)=XI(J)
   11 CONTINUE
AX=0.                      Initial guess for brackets.
XX=1.
CALL MNBRAK(AX,XX,BX,FA,FX,FB,F1DIM)
FRET=BRENT(AX,XX,BX,F1DIM,TOL,XMIN)
DO 12 J=1,N                Construct the vector results to return.
      XI(J)=XMIN*XI(J)
      P(J)=P(J)+XI(J)
   12 CONTINUE
RETURN
END
```

```
FUNCTION F1DIM(X)
    Must accompany LINMIN.
PARAMETER (NMAX=50)
COMMON /F1COM/ NCOM,PCOM(NMAX),XICOM(NMAX)
DIMENSION XT(NMAX)
DO 11 J=1,NCOM
      XT(J)=PCOM(J)+X*XICOM(J)
   11 CONTINUE
F1DIM=FUNC(XT)
RETURN
END
```


REFERENCES AND FURTHER READING:

Brent, Richard P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, N.J.: Prentice-Hall), Chapter 7.

Acton, Forman S. 1970, *Numerical Methods That Work* (New York: Harper and Row), pp. 464–467.

Jacobs, David 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 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 all the way back to Cauchy), is similar to the problem which 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

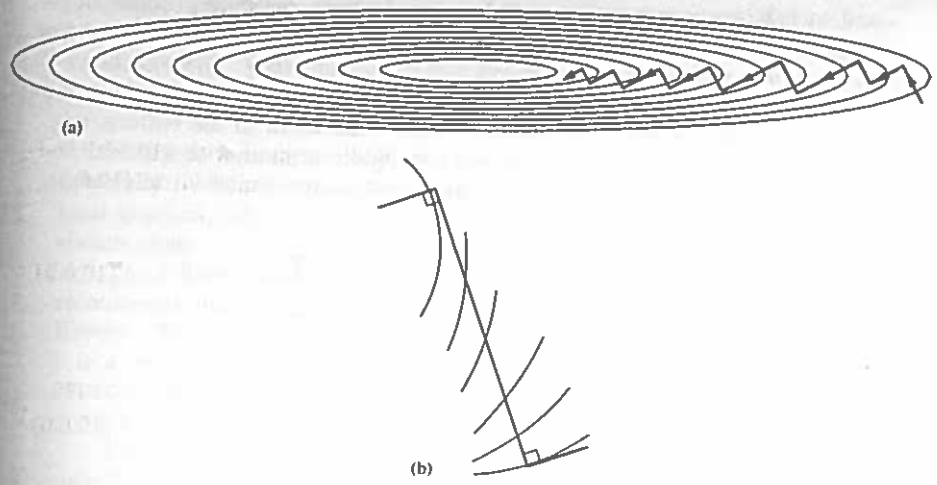

Figure 10.6.1. (a) Steepest descent method in a long, narrow "valley". While more efficient than the strategy of Figure 10.5.1, steepest descent is nonetheless an inefficient strategy, taking many steps to reach the valley floor. (b) Magnified view of one step: a step starts off in the local gradient 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.

possible, to all previous directions traversed. Methods which accomplish this construction are called *conjugate gradient* methods.

The two most important conjugate gradient methods are the *Fletcher-Reeves method* and the *Polak-Ribiere method*. These two methods are closely related. Both are based on the following, rather remarkable, theorem: Let $\mathbf{A}$ be a symmetric, positive-definite, $n \times n$ matrix. Let $\mathbf{g}_0$ be an arbitrary vector; let $\mathbf{h}_0 = \mathbf{g}_0$. For $i = 0, 1, 2, \ldots$ define the two sequences of vectors

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

where λ_i, γ_i are chosen to make $\mathbf{g}_{i+1} \cdot \mathbf{g}_i = 0$ and $\mathbf{h}_{i+1} \cdot \mathbf{A} \cdot \mathbf{h}_i = 0$, i.e.

$$\lambda_i = \frac{\mathbf{g}_i \cdot \mathbf{g}_i}{\mathbf{g}_i \cdot \mathbf{A} \cdot \mathbf{h}_i} \qquad \gamma_i = \frac{\mathbf{g}_{i+1} \cdot \mathbf{A} \cdot \mathbf{h}_i}{\mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_i} \tag{10.6.3}$$

(unless the denominators equal zero, in which case take $\lambda_i = 0$, $\gamma_i = 0$). Then, for *all* $i \neq j$,

$$\mathbf{g}_i \cdot \mathbf{g}_j = 0 \qquad \mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_j = 0 \tag{10.6.4}$$

In other words, the procedure (10.6.2), which is a kind of *Gram-Schmidt bi-orthogonalization* making each $\mathbf{g}_i$ *orthogonal* to its immediate predecessor,

and each $\mathbf{h}_i$ *conjugate* to its immediate predecessor, actually produces a sequence of $\mathbf{g}$'s that are all *mutally* orthogonal, and a sequence of $\mathbf{h}$'s that are all *mutually* conjugate!

The proof of this theorem proceeds by straightforward induction. For details, consult the book by Polak.

Knowing that (10.6.4) holds, you can fiddle around with (10.6.2) and verify that the following expressions for γ_i and λ_i are equivalent to (10.6.3),

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

$$\lambda_i = \frac{\mathbf{g}_i \cdot \mathbf{h}_i}{\mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_i} \tag{10.6.6}$$

We are now ready to apply this formalism to the problem of minimizing a function approximated by a quadratic form (10.6.1). Suppose that we knew the Hessian matrix $\mathbf{A}$. 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 another remarkable theorem to save the day: Let $\mathbf{g}_i$ and $\mathbf{h}_i$ be the same sequence of vectors as before. 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

$$\mathbf{g}_{i+1} = -\mathbf{A} \cdot (\mathbf{P}_i + \lambda \mathbf{h}_i) + \mathbf{b} = \mathbf{g}_i - \lambda \mathbf{A} \cdot \mathbf{h}_i \tag{10.6.7}$$

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 (10.6.7) to solve for λ. The result is exactly the expression (10.6.6). But with this value of λ, (10.6.7) is the same as (10.6.2), q.e.d.

We have, then, the basis of an algorithm which 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.

Thus far, everything said is applicable both to the Fletcher-Reeves and to the Polak-Ribiere methods. There is only one tiny, but sometimes significant, difference between the two methods. Fletcher and Reeves originally used the first expression for γ_i of equation (10.6.5) above. Polak and Ribiere, later, proposed using the second expression in the same equation. "Wait," you say, "aren't they equal?" 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 (see Jacobs) that the Polak-Ribiere formula accomplishes the transition to further iterations more gracefully: When it runs out of steam, it tends to reset $\mathbf{h}$ 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` is a vector of length `N`, and also presumes the existence of a subroutine `DFUNC(P,DF)` that returns the vector gradient `DF` 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` which uses `DBRENT` instead of `BRENT`, i.e., which uses the gradient in doing the line minimizations. See note below.

```
      SUBROUTINE FRPRMN(P,N,FTOL,ITER,FRET)
        Given a starting point P that is a vector of length N, Fletcher-Reeves-Polak-Ribiere min-
        imization 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.
      PARAMETER (NMAX=50,ITMAX=200,EPS=1.E-10)
        Maximum anticipated value of N; maximum allowed number of iterations; small number to
        rectify special case of converging to exactly zero function value.
      DIMENSION P(N),G(NMAX),H(NMAX),XI(NMAX)
      FP=FUNC(P)                          Initializations.
      CALL DFUNC(P,XI)
      DO 11 J=1,N
        G(J)=-XI(J)
        H(J)=G(J)
        XI(J)=H(J)
      11 CONTINUE
      DO 14 ITS=1,ITMAX                   Loop over iterations.
        ITER=ITS
        CALL LINMIN(P,XI,N,FRET)          Next statement is the normal return:
        IF(2.*ABS(FRET-FP).LE.FTOL*(ABS(FRET)+ABS(FP)+EPS))RETURN
        FP=FUNC(P)
        CALL DFUNC(P,XI)
        GG=0.
        DGG=0.
        DO 12 J=1,N
          GG=GG+G(J)**2
C         DGG=DGG+XI(J)**2                This statement for Fletcher-Reeves.
          DGG=DGG+(XI(J)+G(J))*XI(J)      This statement for Polak-Ribiere.
        12 CONTINUE
        IF(GG.EQ.0.)RETURN                Unlikely. If gradient is exactly zero then we are already done.
        GAM=DGG/GG
        DO 13 J=1,N
          G(J)=-XI(J)
          H(J)=G(J)+GAM*H(J)
          XI(J)=H(J)
        13 CONTINUE
```

```
  14 CONTINUE
PAUSE 'FRPR maximum iterations exceeded'
RETURN
END
```

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.

Rather than reprint the whole routine LINMIN just to show one modified statement, let us just tell you what the change is: The statement

```
FRET=BRENT(AX,XX,BX,F1DIM,TOL,XMIN)
```

should be replaced by

```
FRET=DBRENT(AX,XX,BX,F1DIM,DF1DIM,TOL,XMIN)
```

You must also include the following function, which is analogous in function to F1DIM as discussed in §10.5. And remember, your function must be named FUNC, and its gradient calculation must be named DFUNC.

```
FUNCTION DF1DIM(X)
PARAMETER (NMAX=50)
COMMON /F1COM/ NCOM,PCOM(NMAX),XICOM(NMAX)
DIMENSION XT(NMAX),DF(NMAX)
DO 11 J=1,NCOM
  XT(J)=PCOM(J)+X*XICOM(J)
 11 CONTINUE
CALL DFUNC(XT,DF)
DF1DIM=0.
DO 12 J=1,NCOM
  DF1DIM=DF1DIM+DF(J)*XICOM(J)
 12 CONTINUE
RETURN
END
```

Note on Sparse Linear Systems

In §2.10 we gave a routine SPARSE for solving a sparse linear systems of equations

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{10.6.8}$$

by using the conjugate gradient method of minimization. You should now be in a position to understand the inner workings of that program.

It was remarked in §2.10 that high accuracy was difficult to obtain because the condition number of the quadratic form that was minimized was the *square* of the condition number of A. This was the case because A was not known to be symmetric and positive definite, so it had to be "squared" before use.

From this section, you can see that if your matrix A *is* known to be symmetric and positive definite (or *negative* definite, in which case just use $-\mathbf{A}$), then you can use A directly:

To solve (10.6.8), you write subroutines which calculate, for a given x,

$$f(\mathbf{x}) \equiv \frac{1}{2}\mathbf{x} \cdot \mathbf{A} \cdot \mathbf{x} - \mathbf{b} \cdot \mathbf{x} \qquad \nabla f \equiv \mathbf{A} \cdot \mathbf{x} - \mathbf{b} \tag{10.6.9}$$

(compare equations 10.5.1 and 10.5.3). Take care to write these routines as cleverly as you can, taking advantage of the sparseness of your matrix.

Now call FRPRMN with any starting value of x, e.g. $\mathbf{x} = 0$. The value of x returned should be the solution of (10.6.8).

REFERENCES AND FURTHER READING:

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

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

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 an 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.

The most frequently implemented variable metric method is the *Davidon-Fletcher-Powell (DFP)* algorithm (sometimes referred to as simply *Fletcher-Powell*). A related method which goes by the name *Broyden-Fletcher-Goldfarb-Shanno (BFGS)* algorithm also has proved important. Dixon has shown that 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 (see Jacobs). 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 ∞.

If we are successful in achieving (10.7.1), then we can use $\mathbf{H}$ as follows: The minimum point $\mathbf{x}_m$ satisfies

$$\mathbf{A} \cdot \mathbf{x}_m = \mathbf{b} \tag{10.7.2}$$

(compare equation 10.5.3). At the current point $\mathbf{x}_i$, whatever it is, we have

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

(also by 10.5.3). Subtracting these two equations and multiplying by the inverse matrix $\mathbf{A}^{-1}$, we have

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

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 $\mathbf{H} \approx \mathbf{A}^{-1}$.

We won't rigorously derive the DFP algorithm for taking $\mathbf{H}_i$ into $\mathbf{H}_{i+1}$; you can consult Polak for clear derivations. Following Brodlie (in Jacobs), 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

$$\mathbf{x}_{i+1} - \mathbf{x}_i = \mathbf{A}^{-1} \cdot (\nabla f_{i+1} - \nabla f_i) \tag{10.7.5}$$

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 (10.7.5) as if it were actually $\mathbf{A}^{-1}$, that is,

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

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 (10.7.6) must hold! One such way, the *DFP updating formula* is

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

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 10.7.7 does satisfy 10.7.6.)

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

$$\cdots + [(\nabla f_{i+1} - \nabla f_i) \cdot \mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)]\, \mathbf{u} \otimes \mathbf{u} \tag{10.7.8}$$

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

$$\mathbf{u} \equiv \frac{(\mathbf{x}_{i+1} - \mathbf{x}_i)}{(\mathbf{x}_{i+1} - \mathbf{x}_i) \cdot (\nabla f_{i+1} - \nabla f_i)} - \frac{\mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)}{(\nabla f_{i+1} - \nabla f_i) \cdot \mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)} \tag{10.7.9}$$

(You might also verify that this satisfies 10.7.6.)

You will have to take on faith — or else consult Polak for details of — the "deep" result that equation (10.7.7), with or without (10.7.8), does in fact converge to $\mathbf{A}^{-1}$ in N steps, if f is a quadratic form.

Without further ado, we have the following implementation, which uses the line minimization routine LINMIN as in the previous section.

```
      SUBROUTINE DFPMIN(P,N,FTOL,ITER,FRET)
          Given a starting point P 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 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.
      PARAMETER (NMAX=50,ITMAX=200,EPS=1.E-10)
          Maximum anticipated value of N; maximum allowed number of iterations; small number to
          rectify special case of converging to exactly zero function value.
      DIMENSION P(N),HESSIN(NMAX,NMAX),XI(NMAX),G(NMAX),DG(NMAX),HDG(NMAX)
      FP=FUNC(P)                      Calculate starting function value and gradient.
      CALL DFUNC(P,G)
      DO 12 I=1,N                     and initialize the inverse Hessian to the unit matrix.
        DO 11 J=1,N
          HESSIN(I,J)=0.
        11 CONTINUE
        HESSIN(I,I)=1.
        XI(I)=-G(I)                   Initial line direction.
      12 CONTINUE
      DO 24 ITS=1,ITMAX               Main loop over the iterations.
        ITER=ITS
        CALL LINMIN(P,XI,N,FRET)      Next statement is the normal return:
        IF(2.*ABS(FRET-FP).LE.FTOL*(ABS(FRET)+ABS(FP)+EPS))RETURN
        FP=FRET                       Save the old function value
        DO 13 I=1,N                   and the old gradient.
          DG(I)=G(I)
        13 CONTINUE
        FRET=FUNC(P)                  Get new function value and gradient.
        CALL DFUNC(P,G)
        DO 14 I=1,N                   Compute difference of gradients,
          DG(I)=G(I)-DG(I)
        14 CONTINUE
        DO 16 I=1,N                   and difference times current matrix.
          HDG(I)=0.
          DO 15 J=1,N
            HDG(I)=HDG(I)+HESSIN(I,J)*DG(J)
          15 CONTINUE
        16 CONTINUE
        FAC=0.                        Calculate dot products for the denominators.
        FAE=0.
        DO 17 I=1,N
          FAC=FAC+DG(I)*XI(I)
          FAE=FAE+DG(I)*HDG(I)
        17 CONTINUE
        FAC=1./FAC                    and make the denominators multiplicative.
        FAD=1./FAE
        DO 18 I=1,N                   The vector which makes BFGS different from DFP:
          DG(I)=FAC*XI(I)-FAD*HDG(I)
        18 CONTINUE
        DO 21 I=1,N                   The BFGS updating formula:
          DO 19 J=1,N
            HESSIN(I,J)=HESSIN(I,J)+FAC*XI(I)*XI(J)
              -FAD*HDG(I)*HDG(J)+FAE*DG(I)*DG(J)
          19 CONTINUE
        21 CONTINUE
        DO 23 I=1,N                   Now calculate the next direction to go,
          XI(I)=0.
          DO 22 J=1,N
            XI(I)=XI(I)-HESSIN(I,J)*G(J)
          22 CONTINUE
        23 CONTINUE
      24 CONTINUE                     and go back for another iteration.
      PAUSE 'too many iterations in DFPMIN'
      RETURN
      END
```

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 which 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

$$\mathbf{A} \cdot (\mathbf{x}_m - \mathbf{x}_i) = -\nabla f(\mathbf{x}_i) \tag{10.7.10}$$

At first glance this seems like a bad idea, since solving (10.7.10) 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*, somewhat similar to the LU decomposition discussed in Chapter 2 but specialized to the case of symmetric, positive-definite matrices. The updating formula for the Cholesky decomposition of $\mathbf{A}$ is such as 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 (see Jacobs).

Another issue is the question of how exact the line searches need to be in order to retain good convergence for the overall method. It seems wasteful to converge so accurately on each line minimization to a way-point that has no particular significance to the N-dimensional function. There are some known results in this area, but beyond our present scope. One useful fact is that the BFGS updating formula is more tolerant of inexactitude in the line minimization than is the DFP updating formula.

Oddly, there seems to be no known minimization method which makes efficient use of *all* the function and/or gradient evaluations which are performed during each line minimization. One might hope for future progress in this area.

REFERENCES AND FURTHER READING:

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

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

Acton, Forman S. 1970, *Numerical Methods That Work* (New York: Harper and Row), pp. 467–468 (note, however, missing minus sign in the DFP algorithm).

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

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

m_2 of them of the form

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

and m_3 of them of the form

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

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 (1981).

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

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

with all the x's non-negative and also with

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

The answer turns out to be (to 2 significant figures) $x_1 = 0$, $x_2 = 3.33$, $x_3 = 4.73$, $x_4 = 0.95$. 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.

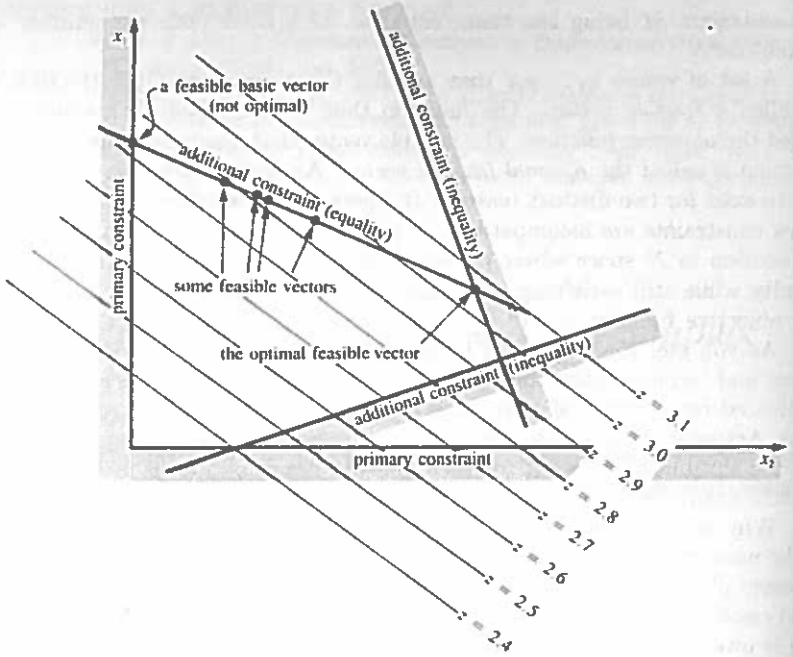

Figure 10.8.1. Basic concepts of linear programming. The case of only two independent variables, x_1, x_2, is shown. The linear function z, to be maximized, is represented by its contour lines. Primary constraints require x_1 and x_2 to be positive. Additional constraints may restrict the solution to 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.

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 non-allowed 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 which are feasible vectors and which satisfy this property, that they 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 which 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, 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 the Kolata reference.)

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 only be achieved 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 non-negative 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 which were 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 first solve 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,

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

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 which 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 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 which 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 which 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

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

(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):

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

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*

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

(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	...
z	17.03	−.95	−.05	−1.05	...
x_2	3.33	−.35	−.15	.35	...
x_3	4.73	−.55	.05	−.45	...
x_4	.95	−.10	.10	.10	...
y_1	730.55	.10	−.10	.90	...

(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 which are not left-hand variables in the output tableau have zero values. Slack variables with zero values represent constraints which are satisfied as equalities.

Routine Implementing the Simplex Method

The following routine is based algorithmically on the implementation of Kuenzi, Tzschach, and Zehnder. 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 first $M + 1$ rows and $N + 1$ columns of A. 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 physical dimensions of A,

```
DIMENSION A(MP,NP)
```
(10.8.20)

must have NP$\geq N + 1$ and MP$\geq M + 2$. 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 returned as 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) which 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 Wilkinson and Reinsch.

```
      SUBROUTINE SIMPLX(A,M,N,MP,NP,M1,M2,M3,ICASE,IZROV,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.
      PARAMETER(MMAX=100,EPS=1.E-6)          MMAX is the maximum number of constraints expected.
      DIMENSION A(MP,NP),IZROV(N),IPOSV(M),L1(MMAX),L2(MMAX),L3(MMAX)
      IF(M.NE.M1+M2+M3)PAUSE 'Bad input constraint counts.'
      NL1=N                                  Initially make all variables right-hand.
      DO 11 K=1,N                            Initialize index lists.
        L1(K)=K
        IZROV(K)=K
11    CONTINUE
      NL2=N                                  Make all artificial variables left-hand,
      DO 12 I=1,M                            and initialize those lists.
        IF(A(I+1,1).LT.0.)PAUSE 'Bad input tableau.'     Constants b_i must be nonnegative.
        L2(I)=I
        IPOSV(I)=N+I
12    CONTINUE
      DO 13 I=1,M2                           Used later, but initialized here.
        L3(I)=1
13    CONTINUE
      IR=0                                   This flag setting means we are in phase two, i.e have a feasible
      IF(M2+M3.EQ.0)GO TO 30                       starting solution. GOTO 30 if origin is a feasible solution.
      IR=1                                   Flag meaning that we must start out in phase one.
      DO 15 K=1,N+1                          Compute the auxiliary objective function.
        Q1=0.
        DO 14 I=M1+1,M
          Q1=Q1+A(I+1,K)
14      CONTINUE
        A(M+2,K)=-Q1
15    CONTINUE
10    CALL SIMP1(A,MP,NP,M+1,L1,NL1,0,KP,BMAX)     Find max. coeff. of auxiliary objective fn.
      IF(BMAX.LE.EPS.AND.A(M+2,1).LT.-EPS)THEN
        ICASE=-1                             Auxiliary objective function is still negative and can't be improved,
        RETURN                                     hence no feasible solution exists.
      ELSE IF(BMAX.LE.EPS.AND.A(M+2,1).LE.EPS)THEN
        M12=M1+M2+1                          Auxiliary objective function is zero and can't be improved. This sig-
        IF(M12.LE.M)THEN                           nals that we have a feasible starting vector. Clean out the
                                                   artificial variables by GOTO 1's and then move on to phase two
```



```
          DO 16 IP=M12,M
            IF(IPOSV(IP).EQ.IP+N)THEN
              CALL SIMP1(A,MP,NP,IP,L1,NL1,1,KP,BMAX)
              IF(BMAX.GT.0.)GO TO 1
            ENDIF
16        CONTINUE
        ENDIF
        IR=0                                 Set flag indicating we have reached phase two.
        M12=M12-1
        IF(M1+1.GT.M12)GO TO 30
        DO 18 I=M1+1,M12
          IF(L3(I-M1).EQ.1)THEN
            DO 17 K=1,N+1
              A(I+1,K)=-A(I+1,K)
17          CONTINUE
          ENDIF
18      CONTINUE
        GO TO 30                             Go to phase two.
      ENDIF
      CALL SIMP2(A,M,N,MP,NP,L2,NL2,IP,KP,Q1)      Locate a pivot element (phase one).
      IF(IP.EQ.0)THEN                        Maximum of auxiliary objective function is unbounded, so no feasible
        ICASE=-1                                   solution exists.
        RETURN
      ENDIF
1     CALL SIMP3(A,MP,NP,M+1,N,IP,KP)              Exchange a left- and a right-hand variable (phase one), then
      IF(IPOSV(IP).GE.N+M1+M2+1)THEN               update lists.
        DO 19 K=1,NL1
          IF(L1(K).EQ.KP)GO TO 2
19      CONTINUE
2       NL1=NL1-1
        DO 21 IS=K,NL1
          L1(IS)=L1(IS+1)
21      CONTINUE
      ELSE
        IF(IPOSV(IP).LT.N+M1+1)GO TO 20
        KH=IPOSV(IP)-M1-N
        IF(L3(KH).EQ.0)GO TO 20
        L3(KH)=0
      ENDIF
      A(M+2,KP+1)=A(M+2,KP+1)+1.
      DO 22 I=1,M+2
        A(I,KP+1)=-A(I,KP+1)
22    CONTINUE
20    IS=IZROV(KP)
      IZROV(KP)=IPOSV(IP)
      IPOSV(IP)=IS
      IF(IR.NE.0)GO TO 10                    If still in phase one, go back to 10.
        End of phase one code for finding an initial feasible solution. Now, in phase two, optimize
        it.
30    CALL SIMP1(A,MP,NP,0,L1,NL1,0,KP,BMAX)  Test the z-row for doneness.
      IF(BMAX.LE.0.)THEN                     Done. Solution found. Return with the good news.
        ICASE=0
        RETURN
      ENDIF
      CALL SIMP2(A,M,N,MP,NP,L2,NL2,IP,KP,Q1)      Locate a pivot element (phase two).
      IF(IP.EQ.0)THEN                        Objective function is unbounded. Report and return.
        ICASE=1
        RETURN
      ENDIF
      CALL SIMP3(A,MP,NP,M,N,IP,KP)                Exchange a left- and a right-hand variable (phase two),
      GO TO 20                               and return for another iteration.
      END
```

The preceding routine makes use of the following utility subroutines.

```
      SUBROUTINE SIMP1(A,MP,NP,MM,LL,NLL,IABF,KP,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.
      DIMENSION A(MP,NP),LL(NP)
      KP=LL(1)
      BMAX=A(MM+1,KP+1)
      IF(NLL.LT.2)RETURN
      DO 11 K=2,NLL
        IF(IABF.EQ.0)THEN
          TEST=A(MM+1,LL(K)+1)-BMAX
        ELSE
          TEST=ABS(A(MM+1,LL(K)+1))-ABS(BMAX)
        ENDIF
        IF(TEST.GT.0.)THEN
          BMAX=A(MM+1,LL(K)+1)
          KP=LL(K)
        ENDIF
      11 CONTINUE
      RETURN
      END
```

```
      SUBROUTINE SIMP2(A,M,N,MP,NP,L2,NL2,IP,KP,Q1)
        Locate a pivot element, taking degeneracy into account.
      PARAMETER (EPS=1.E-6)
      DIMENSION A(MP,NP),L2(MP)
      IP=0
      IF(NL2.LT.1)RETURN
      DO 11 I=1,NL2
        IF(A(L2(I)+1,KP+1).LT.-EPS)GO TO 2
      11 CONTINUE
      RETURN                          No possible pivots.  Return with message.
    2 Q1=-A(L2(I)+1,1)/A(L2(I)+1,KP+1)
      IP=L2(I)
      IF(I+1.GT.NL2)RETURN
      DO 13 I=I+1,NL2
        II=L2(I)
        IF(A(II+1,KP+1).LT.-EPS)THEN
          Q=-A(II+1,1)/A(II+1,KP+1)
          IF(Q.LT.Q1)THEN
            IP=II
            Q1=Q
          ELSE IF (Q.EQ.Q1) THEN              We have a degeneracy.
            DO 12 K=1,N
              QP=-A(IP+1,K+1)/A(IP+1,KP+1)
              QO=-A(II+1,K+1)/A(II+1,KP+1)
              IF(QO.NE.QP)GO TO 6
            12 CONTINUE
    6       IF(QO.LT.QP)IP=II
          ENDIF
        ENDIF
      13 CONTINUE
      RETURN
      END
```

```
      SUBROUTINE SIMP3(A,MP,NP,I1,K1,IP,KP)
        Matrix operations to exchange a left-hand and right-hand variable (see text).
      DIMENSION A(MP,NP)
      PIV=1./A(IP+1,KP+1)
      IF(I1.GE.0)THEN
        DO 12 II=1,I1+1
          IF(II-1.NE.IP)THEN
            A(II,KP+1)=A(II,KP+1)*PIV
            DO 11 KK=1,K1+1
              IF(KK-1.NE.KP)THEN
                A(II,KK)=A(II,KK)-A(IP+1,KK)*A(II,KP+1)
              ENDIF
            11 CONTINUE
          ENDIF
        12 CONTINUE
      ENDIF
      DO 13 KK=1,K1+1
        IF(KK-1.NE.KP)A(IP+1,KK)=-A(IP+1,KK)*PIV
      13 CONTINUE
      A(IP+1,KP+1)=PIV
      RETURN
      END
```

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.

REFERENCES AND FURTHER READING:

Bland, R.G. 1981, *Scientific American*, vol. 244, (June) p. 126.

Kolata, G. 1982, *Science*, vol. 217, p. 39.

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

Dantzig, G.B. 1963, *Linear Programming and Extensions* (Princeton, N.J.: Princeton University Press).

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).

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).

10.9 Combinatorial Minimization: Method of Simulated Annealing

The *method of simulated annealing* is a technique that has recently attracted significant attention as suitable for optimization problems of very large scale. For practical purposes, it 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. 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. Amazingly, the implementation of the algorithm is quite 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" 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.

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 which 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 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.

The Traveling Salesman Problem

A concrete illustration is provided by the traveling salesman problem. The salesperson 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(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. 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,

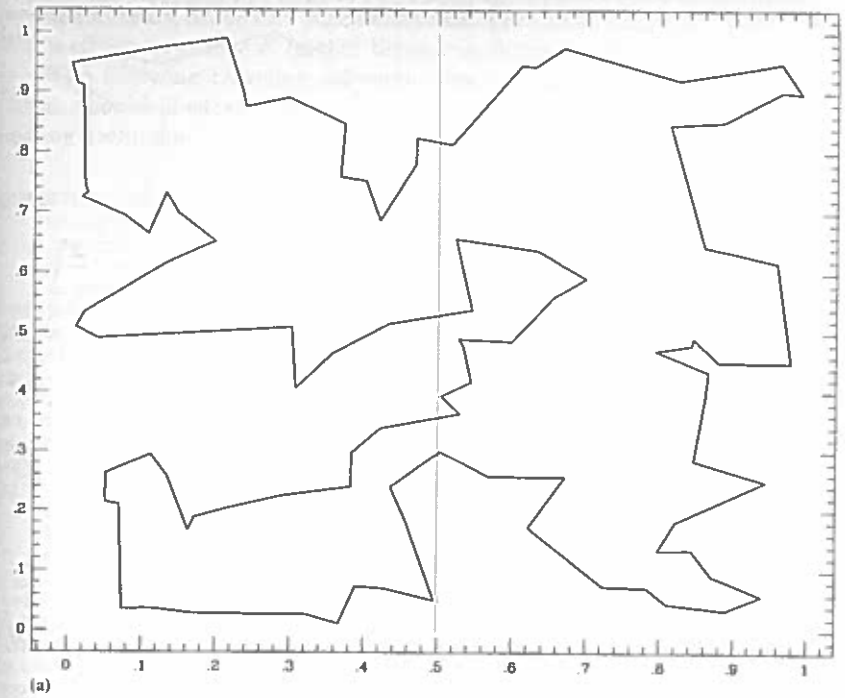

Figure 10.9.1. Traveling salesman problem solved by simulated annealing. The (nearly) shortest path among 100 randomly positioned cities is shown in (a). The dotted line is a river, but there is no penalty in crossing. In (b) the river-crossing penalty is made large, and the solution restricts itself 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!

equal to $+1$ if it is east of the Mississippi, -1 if it is west, and take the objective function to be

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

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 enountered, 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, $100N$

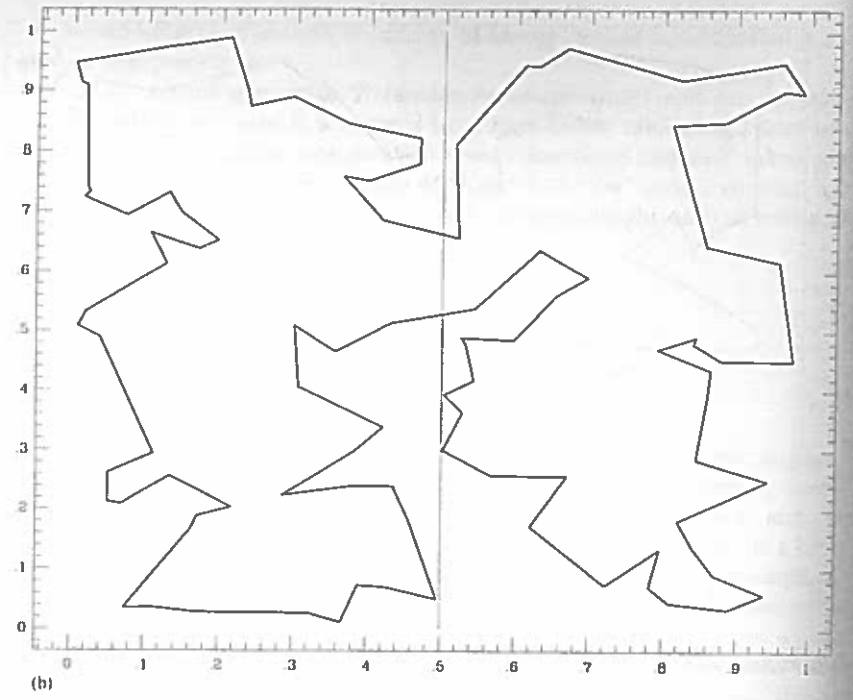

Figure 10.9.1. Traveling salesman problem solved by simulated annealing (see caption on previous page).

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, should illustrate for you the important aspects of the simulated annealing technique.

```
      SUBROUTINE ANNEAL(X,Y,IORDER,NCITY)
        This algorithm finds the shortest round-trip path to NCITY cities whose coordinates are
        in the arrays X(I),Y(I). The array IORDER(I) 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.
      DIMENSION X(NCITY),Y(NCITY),IORDER(NCITY),N(6)
      LOGICAL ANS
      ALEN(X1,X2,Y1,Y2)=SQRT((X2-X1)**2+(Y2-Y1)**2)
      MOVER=100*NCITY                  Maximum number of paths tried at any temperature.
      NLIMIT=10*NCITY                  Maximum number of successful path changes before continuing.
      TFACTR=0.9                       Annealing schedule - T is reduced by this factor on each step.
      PATH=0.0
      T=0.5
      DO 11 I=1,NCITY-1                Calculate initial path length.
        I1=IORDER(I)
        I2=IORDER(I+1)
        PATH=PATH+ALEN(X(I1),X(I2),Y(I1),Y(I2))
   11 CONTINUE
      I1=IORDER(NCITY)                 Close the loop by tying path ends together.
      I2=IORDER(1)
      PATH=PATH+ALEN(X(I1),X(I2),Y(I1),Y(I2))
      IDUM=-1
      ISEED=111
      DO 13 J=1,100                    Try up to 100 temperature steps.
        NSUCC=0
        DO 12 K=1,MOVER
          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).GE.N(1)) N(2)=N(2)+1
          NN=1+MOD((N(1)-N(2)+NCITY-1),NCITY)  NN is the number of cities not on the segment.
          IF (NN.LT.3) GOTO 1
          IDEC=IRBIT1(ISEED)    Decide whether to do a segment reversal or transport.
          IF (IDEC.EQ.0) THEN   Do a transport.
            N(3)=N(2)+INT(ABS(NN-2)*RAN3(IDUM))+1
            N(3)=1+MOD(N(3)-1,NCITY)      Transport to a location not on the path.
            CALL TRNCST(X,Y,IORDER,NCITY,N,DE)   Calculate cost.
            CALL METROP(DE,T,ANS)     Consult the oracle.
            IF (ANS) THEN
              NSUCC=NSUCC+1
              PATH=PATH+DE
              CALL TRNSPT(IORDER,NCITY,N)  Carry out the transport.
            ENDIF
          ELSE                  Do a path reversal
            CALL REVCST(X,Y,IORDER,NCITY,N,DE)   Calculate cost.
            CALL METROP(DE,T,ANS)     Consult the oracle.
            IF (ANS) THEN
              NSUCC=NSUCC+1
              PATH=PATH+DE
              CALL REVERS(IORDER,NCITY,N)  Carry out the reversal.
            ENDIF
          ENDIF
          IF (NSUCC.GE.NLIMIT) GOTO 2  Finish early if we have enough successful changes.
   12   CONTINUE
        WRITE(*,*)
        WRITE(*,*) 'T =',T,' Path Length =',PATH
        WRITE(*,*) 'Successful Moves:  ',NSUCC
```

```
      T=T*TFACTR                Annealing schedule.
      IF (NSUCC.EQ.0) RETURN    If no success, we are done.
    13CONTINUE
RETURN
END
```

```
SUBROUTINE REVCST(X,Y,IORDER,NCITY,N,DE)
    This subroutine returns the value of the cost function for a proposed path reversal.
    NCITY is the number of cities, and arrays X(I),Y(I) give the coordinates of these cities.
    IORDER(I) 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.
DIMENSION X(NCITY),Y(NCITY),IORDER(NCITY),N(6),XX(4),YY(4)
ALEN(X1,X2,Y1,Y2)=SQRT((X2-X1)**2+(Y2-Y1)**2)
N(3)=1+MOD((N(1)+NCITY-2),NCITY)      Find the city before N(1) ..
N(4)=1+MOD(N(2),NCITY)                .. and the city after N(2)
DO11 J=1,4
    II=IORDER(N(J))                   Find coordinates for the four cities involved.
    XX(J)=X(II)
    YY(J)=Y(II)
  11CONTINUE
DE=-ALEN(XX(1),XX(3),YY(1),YY(3))     Calculate cost of disconnecting the segment at both ends
     -ALEN(XX(2),XX(4),YY(2),YY(4))     and reconnecting in the opposite order.
     +ALEN(XX(1),XX(4),YY(1),YY(4))
     +ALEN(XX(2),XX(3),YY(2),YY(3))
RETURN
END
```

```
SUBROUTINE REVERS(IORDER,NCITY,N)
    This routine performs a path segment reversal. IORDER(I) 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) which
    immediately precede and follow this segment. N(3) and N(4) are found by subroutine
    REVCST. On output, IORDER(I) contains the segment from N(1) to N(2) in reversed order.
DIMENSION IORDER(NCITY),N(6)
NN=(1+MOD(N(2)-N(1)+NCITY,NCITY))/2  This many cities must be swapped to effect the reversal.
DO11 J=1,NN
    K=1+MOD((N(1)+J-2),NCITY)     Start at the ends of the segment and swap pairs of cities, moving
    L=1+MOD((N(2)-J+NCITY),NCITY) toward the center.
    ITMP=IORDER(K)
    IORDER(K)=IORDER(L)
    IORDER(L)=ITMP
  11CONTINUE
RETURN
END
```

```
SUBROUTINE TRNCST(X,Y,IORDER,NCITY,N,DE)
    This subroutine returns the value of the cost function for a proposed path segment trans-
    port. NCITY is the number of cities, and arrays X(I) and Y(I) give the city coordinates.
    IORDER 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.
DIMENSION X(NCITY),Y(NCITY),IORDER(NCITY),N(6),XX(6),YY(6)
ALEN(X1,X2,Y1,Y2)=SQRT((X2-X1)**2+(Y2-Y1)**2)
N(4)=1+MOD(N(3),NCITY)          Find the city following N(3)..
N(5)=1+MOD((N(1)+NCITY-2),NCITY)..and the one preceding N(1)..
N(6)=1+MOD(N(2),NCITY)          ..and the one following N(2).
DO11 J=1,6
    II=IORDER(N(J))             Determine coordinates for the six cities involved.
    XX(J)=X(II)
```



```
    YY(J)=Y(II)
  11CONTINUE
DE=-ALEN(XX(2),XX(6),YY(2),YY(6))     Calculate the cost of disconnecting the path segment from
     -ALEN(XX(1),XX(5),YY(1),YY(5))     N(1) to N(2), opening a space between N(3) and N(4),
     -ALEN(XX(3),XX(4),YY(3),YY(4))     connecting the segment in the space, and connecting
     +ALEN(XX(1),XX(3),YY(1),YY(3))     N(5) to N(6).
     +ALEN(XX(2),XX(4),YY(2),YY(4))
     +ALEN(XX(5),XX(6),YY(5),YY(6))
RETURN
END
```

```
SUBROUTINE TRNSPT(IORDER,NCITY,N)
    This routine does the actual path transport, once METROP has approved. IORDER is an input
    array of length NCITY 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) which
    precede and follow the path. N(4), N(5) and N(6) are calculated by subroutine TRNCST.
    On output, IORDER is modified to reflect the movement of the path segment.
PARAMETER(MXCITY=1000)          Maximum number of cities anticipated.
DIMENSION IORDER(NCITY),JORDER(MXCITY),N(6)
M1=1+MOD((N(2)-N(1)+NCITY),NCITY)     Find the number of cities from N(1) to N(2) ..
M2=1+MOD((N(5)-N(4)+NCITY),NCITY)     ..and the number from N(4) to N(5) ..
M3=1+MOD((N(3)-N(6)+NCITY),NCITY)     ..and the number from N(6) to N(3).
NN=1
DO11 J=1,M1
    JJ=1+MOD((J+N(1)-2),NCITY)        Copy the chosen segment.
    JORDER(NN)=IORDER(JJ)
    NN=NN+1
  11CONTINUE
IF (M2.GT.0) THEN
    DO12 J=1,M2                       Then copy the segment from N(4) to N(5).
      JJ=1+MOD((J+N(4)-2),NCITY)
      JORDER(NN)=IORDER(JJ)
      NN=NN+1
    12CONTINUE
ENDIF
IF (M3.GT.0) THEN
    DO13 J=1,M3                       Finally, the segment from N(6) to N(3).
      JJ=1+MOD((J+N(6)-2),NCITY)
      JORDER(NN)=IORDER(JJ)
      NN=NN+1
    13CONTINUE
ENDIF
DO14 J=1,NCITY
    IORDER(J)=JORDER(J)               Copy JORDER back into IORDER.
  14CONTINUE
RETURN
END
```

```
SUBROUTINE METROP(DE,T,ANS)
    Metropolis algorithm. ANS is a logical variable which issues a verdict on whether to
    accept a reconfiguration which leads to a change DE in the objective function E. If DE<0,
    ANS=.TRUE., while if DE>0, ANS is only .TRUE. with probability exp(-DE/T), where T is a
    temperature determined by the annealing schedule.
PARAMETER(JDUM=1)
LOGICAL ANS
ANS=(DE.LT.0.0).OR.(RAN3(JDUM).LT.EXP(-DE/T))
RETURN
END
```

There is not yet enough practical experience with the method of simulated annealing to say definitively that it will realize its current promise. 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 which 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 toward an acceptable solution. Information on this subject is found in the references by Kirkpatrick et al.

REFERENCES AND FURTHER READING:

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

Kirkpatrick, S. 1984, *Journal of Statistical Physics*, vol. 34, p. 975.

Vecchi, M.P. and Kirkpatrick, S. 1983, *IEEE Transactions on Computer Aided Design*, vol. CAD-2, p. 215.

Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller A., and Teller, E. 1953 *J. Chem. Phys.*, vol. 21, p. 1087.

Lin, S. 1965, *Bell Syst. Tech. Journ.*, vol. 44, p. 2245.

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].

Chapter 11. Eigensystems

11.0 Introduction

An $N \times N$ matrix $\mathbf{A}$ is said to have an *eigenvector* $\mathbf{x}$ and corresponding *eigenvalue* λ if

$$\mathbf{A} \cdot \mathbf{x} = \lambda \mathbf{x} \tag{11.0.1}$$

Obviously any multiple of an eigenvector $\mathbf{x}$ will also be an eigenvector, but we won't consider such multiples as being distinct eigenvectors. (The zero vector is not considered to be an eigenvector at all.) Evidently (11.0.1) can hold only if

$$\det |\mathbf{A} - \lambda \mathbf{1}| = 0 \tag{11.0.2}$$

which, if expanded out, is an N^{th} degree polynomial in λ whose roots are the eigenvalues. This proves that there are always N (not necessarily distinct) eigenvalues. Equal eigenvalues coming from multiple roots are called *degenerate*. Root-searching in the characteristic equation (11.0.2) is usually a very poor computational method for finding eigenvalues. We will learn much better ways in this chapter, as well as efficient ways for finding corresponding eigenvectors.

The above two equations also prove that every one of the N eigenvalues has a (not necessarily distinct) corresponding eigenvector: If λ is set to an eigenvalue, then the matrix $\mathbf{A} - \lambda \mathbf{1}$ is singular, and we know that every singular matrix has at least one nonzero vector in its nullspace (see §2.9 on singular value decomposition).

If you add $\tau \mathbf{x}$ to both sides of (11.0.1), you will easily see that the eigenvalues of any matrix can be changed or *shifted* by an additive constant τ by adding to the matrix that constant times the identity matrix. The eigenvectors are unchanged by this shift. Shifting, as we will see, is an important part of many algorithms for computing eigenvalues. We see also that there is no special significance to a zero eigenvalue. Any eigenvalue can be shifted to zero, or any zero eigenvalue can be shifted away from zero.