```
ytry=funk(ptry)              Evaluate the function at the trial point.
if (ytry.lt.y(ihi)) then     If it's better than the highest, then replace the highest.
    y(ihi)=ytry
    do 12 j=1,ndim
        psum(j)=psum(j)-p(ihi,j)+ptry(j)
        p(ihi,j)=ptry(j)
    enddo 12
endif
amotry=ytry
return
END
```

CITED REFERENCES AND FURTHER READING:

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

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

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

10.5 Direction Set (Powell's) Methods in Multidimensions

We know (§10.1–§10.3) how to minimize a function of one variable. If we start at a point **P** in N-dimensional space, and proceed from there in some vector direction **n**, then any function of N variables $f(\mathbf{P})$ can be minimized along the line **n** by our one-dimensional methods. One can dream up various multidimensional minimization methods that consist of sequences of such line minimizations. Different methods will differ only by how, at each stage, they choose the next direction **n** to try. All such methods presume the existence of a "black-box" sub-algorithm, which we might call `linmin` (given as an explicit routine at the end of this section), whose definition can be taken for now as

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

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

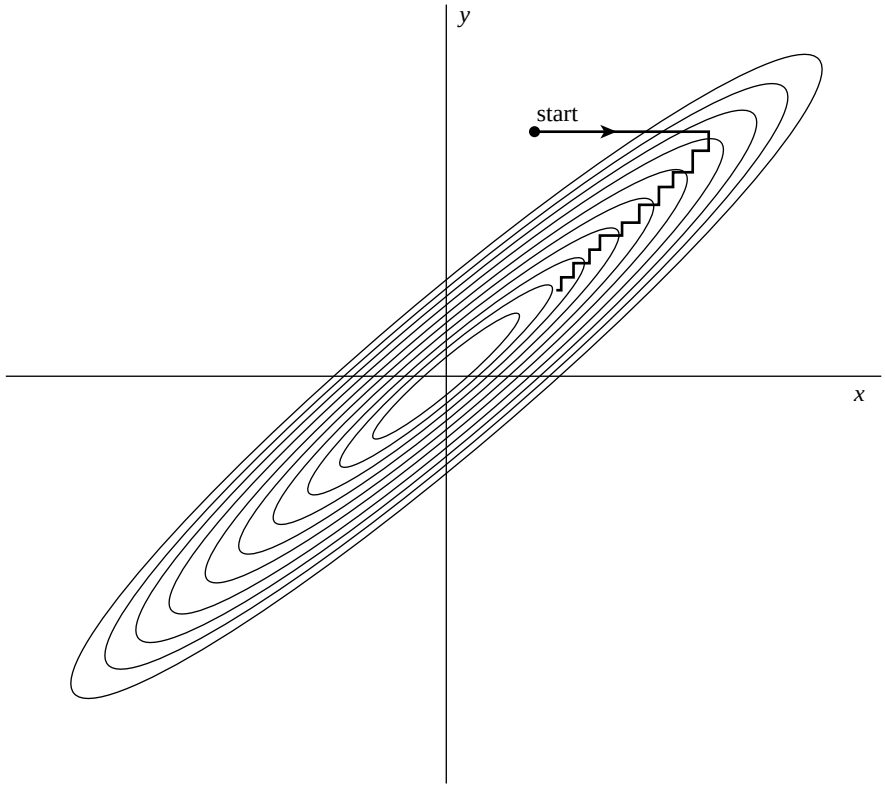

Figure 10.5.1. Successive minimizations along coordinate directions in a long, narrow "valley" (shown as contour lines). Unless the valley is optimally oriented, this method is extremely inefficient, taking many tiny steps to get to the minimum, crossing and re-crossing the principal axis.

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

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

Obviously what we need is a better set of directions than the $\mathbf{e}_i$'s. All *direction set methods* consist of prescriptions for updating the set of directions as the method proceeds, attempting to come up with a set which either (i) includes some very good directions that will take us far along narrow valleys, or else (more subtly)

(ii) includes some number of "non-interfering" directions with the special property that minimization along one is not "spoiled" by subsequent minimization along another, so that interminable cycling through the set of directions can be avoided.

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 **u**, then the gradient of the function must be perpendicular to **u** at the line minimum; if not, then there would still be a nonzero directional derivative along **u**.

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

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

where

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

The matrix **A** whose components are the second partial derivative matrix of the function is called the *Hessian matrix* of the function at **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 **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 **u** to a minimum and now propose to move along some new direction **v**. The condition that motion along **v** not *spoil* our minimization along **u** is just that the gradient stay perpendicular to **u**, i.e., that the change in the gradient be perpendicular to **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 **u** and **v**, they are said to be *conjugate*. When the relation holds pairwise for all members of a set of vectors, they are said to be a conjugate set. If you do successive line minimization of a function along a conjugate set of directions, then you don't need to redo any of those directions (unless, of course, you spoil things by minimizing along a direction that they are *not* conjugate to).

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

Powell's Quadratically Convergent Method

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

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

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

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

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

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

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

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

2. Brent points out that the set of directions can equally well be reset to the columns of any orthogonal matrix. Rather than throw away the information on conjugate directions already built up, he resets the direction set to calculated principal directions of the matrix $\mathbf{A}$ (which he gives a procedure for determining). The calculation is essentially a singular value decomposition algorithm (see §2.6). Brent has a number of other cute tricks up his sleeve, and his modification of

Powell's method is probably the best presently known. Consult [1] for a detailed description and listing of the program. Unfortunately it is rather too elaborate for us to include here.

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

Discarding the Direction of Largest Decrease

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

Sooner or later, however, we do arrive at an approximately ellipsoidal minimum (cf. equation 10.5.1 when **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)
      INTEGER iter,n,np,NMAX,ITMAX
      REAL fret,ftol,p(np),xi(np,np),func
      EXTERNAL func
      PARAMETER (NMAX=20,ITMAX=200)
C     USES func,linmin
        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(1:n); an initial matrix xi(1:n,1:n) with 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.
        Parameters: Maximum expected value of n, and maximum allowed iterations.
      INTEGER i,ibig,j
      REAL del,fp,fptt,t,pt(NMAX),ptt(NMAX),xit(NMAX)
      fret=func(p)
      do 11 j=1,n                          Save the initial point.
        pt(j)=p(j)
      enddo 11
      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)
        enddo 12
        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
      enddo 13
      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. Save the old starting point.
        ptt(j)=2.*p(j)-pt(j)
        xit(j)=p(j)-pt(j)
        pt(j)=p(j)
      enddo 14
      fptt=func(ptt)                       Function value at extrapolated point.
      if(fptt.ge.fp)goto 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.)goto 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)=xi(j,n)
        xi(j,n)=xit(j)
      enddo 15
      goto 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. 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 receives 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.

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)
      INTEGER n,NMAX
      REAL fret,p(n),xi(n),TOL
      PARAMETER (NMAX=50,TOL=1.e-4)          Maximum anticipated n, and TOL passed to brent.
C     USES brent,f1dim,mnbrak
        Given an n-dimensional point p(1:n) and an n-dimensional direction xi(1:n), moves and
        resets p to where the function func(p) takes on a minimum along the direction xi from
        p, and replaces xi by the actual vector displacement that p was moved. Also returns as
        fret the value of func at the returned location p. This is actually all accomplished by
        calling the routines mnbrak and brent.
      INTEGER j,ncom
      REAL ax,bx,fa,fb,fx,xmin,xx,pcom(NMAX),xicom(NMAX),brent
      COMMON /f1com/ pcom,xicom,ncom
      EXTERNAL f1dim
      ncom=n                                 Set up the common block.
      do 11 j=1,n
        pcom(j)=p(j)
        xicom(j)=xi(j)
      enddo 11
      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)
      enddo 12
      return
```

```
      END


      FUNCTION f1dim(x)
      INTEGER NMAX
      REAL f1dim,func,x
      PARAMETER (NMAX=50)
C     USES func
        Used by linmin as the function passed to mnbrak and brent.
      INTEGER j,ncom
      REAL pcom(NMAX),xicom(NMAX),xt(NMAX)
      COMMON /f1com/ pcom,xicom,ncom
      do 11 j=1,ncom
        xt(j)=pcom(j)+x*xicom(j)
      enddo 11
      f1dim=func(xt)
      return
      END
```

CITED REFERENCES AND FURTHER READING:

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

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

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

10.6 Conjugate Gradient Methods in Multidimensions

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

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

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

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

In the direction set methods of §10.5, we collected the necessary information by making on the order of N^2 separate line minimizations, each requiring "a few" (but sometimes a *big* few!) function evaluations. Now, each evaluation of the gradient