10.4 Downhill Simplex Method in Multidimensions

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

The downhill simplex method is due to Nelder and Mead [1]. The method requires only function evaluations, not derivatives. It is not very efficient in terms of the number of function evaluations that it requires. Powell's method (§10.5) is almost surely faster in all likely applications. However, the downhill simplex method may frequently be the best method to use if the figure of merit is "get something working quickly" for a problem whose computational burden is small.

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

A simplex is the geometrical figure consisting, in \( N \) dimensions, of \( N+1 \) points (or vertices) and all their interconnecting line segments, polygonal faces, etc. In two dimensions, a simplex is a triangle. In three dimensions it is a tetrahedron, not necessarily the regular tetrahedron. (The simplex method of linear programming, described in §10.8, also makes use of the geometrical concept of a simplex. Otherwise it is completely unrelated to the algorithm that we are describing in this section.) In general we are only interested in simplexes that are nondegenerate, i.e., that enclose a finite inner \( N \)-dimensional volume. If any point of a nondegenerate simplex is taken as the origin, then the \( N \) other points define vector directions that span the \( N \)-dimensional vector space.

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

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

\[
\mathbf{P}_i = \mathbf{P}_0 + \lambda \mathbf{e}_i
\]

(10.4.1)

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

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

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

Note well that either of the above criteria might be fooled by a single anomalous step that, for one reason or another, failed to get anywhere. Therefore, it is frequently a good idea to restart a multidimensional minimization routine at a point where it claims to have found a minimum. For this restart, you should reinitialize any ancillary input quantities. In the downhill simplex method, for example, you should reinitialize $N$ of the $N+1$ vertices of the simplex again by equation (10.4.1), with $P_0$ 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:

```fortran
SUBROUTINE amoeba(p,y,mp,np,ndim,ftol,funk,iter)
INTEGER iter,mp,ndim,np,NMAX,ITMAX
REAL ftol,p(mp,np),y(mp),funk,TINY
PARAMETER (NMAX=20,ITMAX=5000,TINY=1.e-10)
EXTERNAL funk
C USES amotry,funk
Multidimensional minimization of the function $f(x)$ where $x(1:ndim)$ is a vector in $ndim$ dimensions, by the downhill simplex method of Nelder and Mead. The matrix p(1:ndim+1,1:ndim) is input. Its $ndim+1$ rows are $ndim$-dimensional vectors which are the vertices of the starting simplex. Also input is the vector y(1:ndim+1), whose components must be pre-initialized to the values of $f(x)$ evaluated at the $ndim+1$ vertices (rows) of p; and $f_{tol}$ 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 $f_{tol}$ of a minimum function value, and iter gives the number of function evaluations taken.
INTEGER i,ihi,ilo,inhi,j,m,n
REAL rtol,sum,swap,ysave,ytry,psum(NMAX),amotry
iter=0
1 do i=1,ndim
   sum=0.
   do m=1,ndim+1
      sum=sum+p(m,n)
   enddo
   psum(n)=sum
1 enddo
2 ilo=1
   Enter here when starting or have just overall contracted. 
   Recompute psum.
   Enter here when have just changed a single point.
   Determine which point is the highest (worst), next-highest, and lowest (best).
   by looping over the points in the simplex.
   if(y(i).lt.y(ilo)) ilo=i
   else if(y(i).lt.y(ihi)) then
      inhi=ihi
      ihi=i
   endif
   if(y(i).le.y(ilo)) ili=i
   if(y(i).gt.y(ihi)) then
      inhi=ihi
      ihi=i
   endif
   if(y(i).le.y(inhi)) then
      if(i.ne.ihi) inhi=i
```

Chapter 10. Minimization or Maximization of Functions
10.4 Downhill Simplex Method in Multidimensions

```
ENDIF
END

rtol=2.*abs(y(ihi)-y(ilo))/(abs(y(ihi))+abs(y(ilo))+TINY)
Compute the fractional range from highest to lowest and return if satisfactory.
IF (rtol.gt.ftol) RETURN

swap=y(1)
y(1)=y(ilo)
y(ilo)=swap
DO 14 n=1,ndim
   swap=p(1,n)
p(1,n)=p(ilo,n)
p(ilo,n)=swap
   END DO
14
RETURN

ENDIF
END
```

```
ENDIF
END

DEF=
if (iter.ge.ITMAX) then
   Print 'ITMAX exceeded in amoeba'
   iter=iter+2
   Begin a new iteration. First extrapolate by a factor −1 through the face of the simplex across
   from the high point, i.e., reflect the simplex from the high point.
ytry=amotry(p,y,psum,mp,np,ndim,funk,ihi,-1.0)
Else if (ytry.ge.y(ihi)) then
   The reflected point is worse than the second-highest, so look for an intermediate lower point,
   i.e., do a one-dimensional contraction.
ysave=y(ihi)
ytry=amotry(p,y,psum,mp,np,ndim,funk,ihi,0.5)
Else if (ytry.ge.ysave) then
   Can't seem to get rid of that high point. Better contract
   around the lowest (best) point.
   DO 16 i=1,ndim+1
      IF (i.ne.ilo) then
         DO 15 j=1,ndim
            psum(j)=0.5*(p(i,j)+p(ilo,j))
p(i,j)=psum(j)
         END DO
15
         y(i)=funk(psum)
      ELSE
         y(i)=y(ihi)
      ENDIF
16
   ITER=ITER+NDIM
   KEEP TRACK OF FUNCTION EVALUATIONS.
   GO TO 1
ELSE
   ITER=ITER-1
   CORRECT THE EVALUATION COUNT.
ENDIF
GOTO 2
```

```
FUNCTION amotry(p,y,psum,mp,np,ndim,funk,ihi,fac)
INTEGER ihi,mp,ndim,np,NMAX
REAL amotry,fac,p(mp,np),psum(np),y(mp),funk
PARAMETER (NMAX=20)
EXTERNAL funk
C USES funk
Extrapolates by a factor fac through the face of the simplex across from the high point,
tries it, and replaces the high point if the new point is better.
INTEGER j
REAL fac1,fac2,ytry,try(NMAX)
fac1=1.-fac/ndim
fac2=fac1-fac
DO 11 j=1,ndim
   try(j)=psum(j)*fac1-p(ihi,j)*fac2
   END DO
11
RETURN
```

```
FUNCTION amotry(p,y,psum,mp,np,ndim,funk,ihi,fac)
INTEGER ihi,mp,ndim,np,NMAX
REAL amotry,fac,p(mp,np),psum(np),y(mp),funk
PARAMETER (NMAX=20)
EXTERNAL funk
C USES funk
Extrapolates by a factor fac through the face of the simplex across from the high point,
tries it, and replaces the high point if the new point is better.
INTEGER j
REAL fac1,fac2,ytry,try(NMAX)
fac1=1.-fac/ndim
fac2=fac1-fac
DO 11 j=1,ndim
   try(j)=psum(j)*fac1-p(ihi,j)*fac2
   END DO
11
RETURN
```

```
FUNCTION amotry(p,y,psum,mp,np,ndim,funk,ihi,fac)
INTEGER ihi,mp,ndim,np,NMAX
REAL amotry,fac,p(mp,np),psum(np),y(mp),funk
PARAMETER (NMAX=20)
EXTERNAL funk
C USES funk
Extrapolates by a factor fac through the face of the simplex across from the high point,
tries it, and replaces the high point if the new point is better.
INTEGER j
REAL fac1,fac2,ytry,try(NMAX)
fac1=1.-fac/ndim
fac2=fac1-fac
DO 11 j=1,ndim
   try(j)=psum(j)*fac1-p(ihi,j)*fac2
   END DO
11
RETURN
```

```
FUNCTION amotry(p,y,psum,mp,np,ndim,funk,ihi,fac)
INTEGER ihi,mp,ndim,np,NMAX
REAL amotry,fac,p(mp,np),psum(np),y(mp),funk
PARAMETER (NMAX=20)
EXTERNAL funk
C USES funk
Extrapolates by a factor fac through the face of the simplex across from the high point,
tries it, and replaces the high point if the new point is better.
INTEGER j
REAL fac1,fac2,ytry,try(NMAX)
fac1=1.-fac/ndim
fac2=fac1-fac
DO 11 j=1,ndim
   try(j)=psum(j)*fac1-p(ihi,j)*fac2
   END DO
11
RETURN
```

```
FUNCTION amotry(p,y,psum,mp,np,ndim,funk,ihi,fac)
INTEGER ihi,mp,ndim,np,NMAX
REAL amotry,fac,p(mp,np),psum(np),y(mp),funk
PARAMETER (NMAX=20)
EXTERNAL funk
C USES funk
Extrapolates by a factor fac through the face of the simplex across from the high point,
tries it, and replaces the high point if the new point is better.
INTEGER j
REAL fac1,fac2,ytry,try(NMAX)
fac1=1.-fac/ndim
fac2=fac1-fac
DO 11 j=1,ndim
   try(j)=psum(j)*fac1-p(ihi,j)*fac2
   END DO
11
RETURN
```
Chapter 10. Minimization or Maximization of Functions

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

CITED REFERENCES AND FURTHER READING:

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 \( f(P) \) 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 \( \text{linmin} \) (given as an explicit routine at the end of this section), whose definition can be taken for now as

\[
\text{linmin}: \text{Given as input the vectors } P \text{ and } n, \text{ and the function } f, \text{ find the scalar } \lambda \text{ that minimizes } f(P + \lambda n). \text{ Replace } P \text{ by } P + \lambda n, \text{ and } n \text{ by } \lambda n. \text{ Done.}
\]

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, \( \text{lnsrch} \).) 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 \( \text{linmin} \) 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 \( \text{linmin} \) and \( \text{not} \) use them in the choice of directions, since in this latter role they can drastically reduce the total computational burden.