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

```
#include "nrutil.h"

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

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

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

CITED REFERENCES AND FURTHER READING:

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

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

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

10.7 Variable Metric Methods in Multidimensions

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

Both variable metric and conjugate gradient methods require that you are able to compute your function's gradient, or first partial derivatives, at arbitrary points. The variable metric approach differs from the conjugate gradient in the way that it stores and updates the information that is accumulated. Instead of requiring intermediate storage on the order of N, the number of dimensions, it requires a matrix of size $N \times N$. Generally, for any moderate N, this is an entirely trivial disadvantage.

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

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

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

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

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

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

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

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

so

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

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

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

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

The "quasi" in quasi-Newton is because we don't use the actual Hessian matrix of f, but instead use our current approximation of it. This is often *better* than using the true Hessian. We can understand this paradoxical result by considering the *descent directions* of f at $\mathbf{x}_i$. These are the directions $\mathbf{p}$ along which f decreases: $\nabla f \cdot \mathbf{p} < 0$. For the Newton direction (10.7.4) to be a descent direction, we must have

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(You might also verify that this satisfies 10.7.7.)

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

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

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

void dfpmin(float p[], int n, float gtol, int *iter, float *fret,
	float(*func)(float []), void (*dfunc)(float [], float []))
```

Given a starting point `p[1..n]` that is a vector of length `n`, the Broyden-Fletcher-Goldfarb-Shanno variant of Davidon-Fletcher-Powell minimization is performed on a function `func`, using its gradient as calculated by a routine `dfunc`. The convergence requirement on zeroing the gradient is input as `gtol`. Returned quantities are `p[1..n]` (the location of the minimum), `iter` (the number of iterations that were performed), and `fret` (the minimum value of the function). The routine `lnsrch` is called to perform approximate line minimizations.

```
{
	void lnsrch(int n, float xold[], float fold, float g[], float p[], float x[],
		 float *f, float stpmax, int *check, float (*func)(float []));
	int check,i,its,j;
	float den,fac,fad,fae,fp,stpmax,sum=0.0,sumdg,sumxi,temp,test;
	float *dg,*g,*hdg,**hessin,*pnew,*xi;

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

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

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

Advanced Implementations of Variable Metric Methods

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

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

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

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

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

CITED REFERENCES AND FURTHER READING:

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

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

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

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

10.8 Linear Programming and the Simplex Method

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

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

subject to the primary constraints

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