CITED REFERENCES AND FURTHER READING:

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

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

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

10.6 Conjugate Gradient Methods in Multidimensions

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

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

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

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

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

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

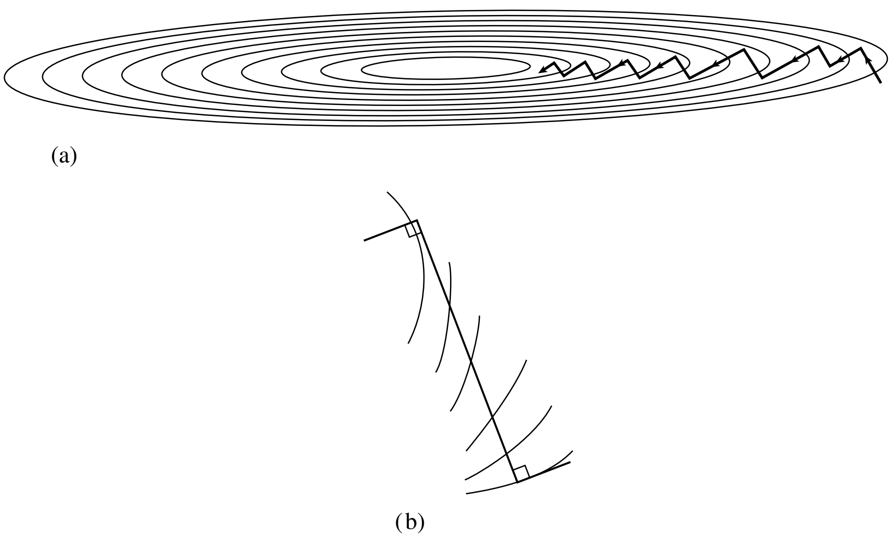

Figure 10.6.1. (a) Steepest descent method in a long, narrow "valley." While more efficient than the strategy of Figure 10.5.1, steepest descent is nonetheless an inefficient strategy, taking many steps to reach the valley floor. (b) Magnified view of one step: A step starts off in the local gradient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

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

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

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

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

In §2.7 we discussed the conjugate gradient method as a technique for solving linear algebraic equations by minimizing a quadratic form. That formalism can also be applied to the problem of minimizing a function *approximated* by the quadratic form (10.6.1). Recall that, starting with an arbitrary initial vector $\mathbf{g}_0$ and letting $\mathbf{h}_0 = \mathbf{g}_0$, the conjugate gradient method constructs two sequences of vectors from the recurrence

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

The vectors satisfy the orthogonality and conjugacy conditions

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

The scalars λ_i and γ_i are given by

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Note on Line Minimization Using Derivatives

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

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

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

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

void dlinmin(float p[], float xi[], int n, float *fret, float (*func)(float []),
    void (*dfunc)(float [], float []))
```

Given an n-dimensional point `p[1..n]` and an n-dimensional direction `xi[1..n]`, moves and resets `p` to where the function `func(p)` takes on a minimum along the direction `xi` from `p`, and replaces `xi` by the actual vector displacement that `p` was moved. Also returns as `fret` the value of `func` at the returned location `p`. This is actually all accomplished by calling the routines `mnbrak` and `dbrent`.

```
{
    float dbrent(float ax, float bx, float cx,
        float (*f)(float), float (*df)(float), float tol, float *xmin);
    float f1dim(float x);
    float df1dim(float x);
    void mnbrak(float *ax, float *bx, float *cx, float *fa, float *fb,
        float *fc, float (*func)(float));
    int j;
    float xx,xmin,fx,fb,fa,bx,ax;

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

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

```
#include "nrutil.h"

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

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

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

CITED REFERENCES AND FURTHER READING:

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

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

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

10.7 Variable Metric Methods in Multidimensions

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

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