Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington, DC: 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.

10.8 Conjugate Gradient Methods in Multidimensions

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 (10.7.1),

$$f(\mathbf{x}) \approx c - \mathbf{b}\cdot\mathbf{x} + \tfrac{1}{2}\,\mathbf{x}\cdot\mathbf{A}\cdot\mathbf{x} \tag{10.8.1}$$

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 §10.7, 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, 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 $-\nabla f(\mathbf{P}_i)$.

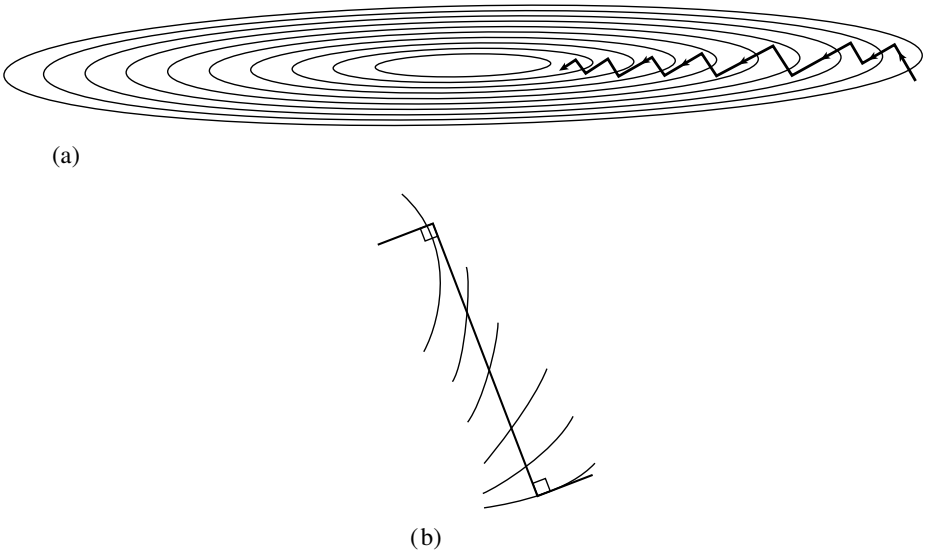

Figure 10.8.1. (a) Steepest descent method in a long, narrow "valley." While more efficient than the strategy of Figure 10.7.1, steepest descent is nonetheless an inefficient strategy, taking many steps to reach the valley floor. (b) Magnified view of one step: A step starts off in the local gradient 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.

The problem with the steepest descent method (which, incidentally, goes back to Cauchy), is similar to the problem that was shown in Figure 10.7.1. 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 10.8.1.)

Just as in the discussion that led up to equation (10.7.5), 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 (10.8.1). 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

$$\mathbf{g}_{i+1} = \mathbf{g}_i - \lambda_i \mathbf{A} \cdot \mathbf{h}_i \qquad \mathbf{h}_{i+1} = \mathbf{g}_{i+1} + \gamma_i \mathbf{h}_i \qquad i = 0, 1, 2, \ldots \tag{10.8.2}$$

The vectors satisfy the orthogonality and conjugacy conditions

$$\mathbf{g}_i \cdot \mathbf{g}_j = 0 \qquad \mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_j = 0 \qquad \mathbf{g}_i \cdot \mathbf{h}_j = 0 \qquad j < i \tag{10.8.3}$$

The scalars λ_i and γ_i are given by

$$\lambda_i = \frac{\mathbf{g}_i \cdot \mathbf{g}_i}{\mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_i} = \frac{\mathbf{g}_i \cdot \mathbf{h}_i}{\mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_i} \tag{10.8.4}$$

$$\gamma_i = \frac{\mathbf{g}_{i+1} \cdot \mathbf{g}_{i+1}}{\mathbf{g}_i \cdot \mathbf{g}_i} \tag{10.8.5}$$

Equations (10.8.2) – (10.8.5) 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 (10.8.1). Then we could use the construction (10.8.2) 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 (10.8.1). Suppose also 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 (10.8.2). (And we have constructed it without knowledge of $\mathbf{A}$!)

Proof: By equation (10.7.3), $\mathbf{g}_i = -\mathbf{A} \cdot \mathbf{P}_i + \mathbf{b}$ and

$$\mathbf{g}_{i+1} = -\mathbf{A} \cdot (\mathbf{P}_i + \lambda \mathbf{h}_i) + \mathbf{b} = \mathbf{g}_i - \lambda \mathbf{A} \cdot \mathbf{h}_i \tag{10.8.6}$$

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 (10.8.6) to solve for λ. The result is exactly the expression (10.8.4). But with this value of λ, (10.8.6) is the same as (10.8.2), 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

$$\gamma_i = \frac{(\mathbf{g}_{i+1} - \mathbf{g}_i) \cdot \mathbf{g}_{i+1}}{\mathbf{g}_i \cdot \mathbf{g}_i} \tag{10.8.7}$$

instead of equation (10.8.5). "Wait," you say, "aren't they equal by the orthogonality conditions (10.8.3)?" 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 $\mathbf{h}$ 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 functor (not a function) that returns the function value by overloading `operator()` and also provides a function to set the vector gradient `df[0..n-1]` evaluated at the input point `p`. Here's an example for the function $x_0^2 + x_1^2$:

```
struct Funcd {                                  Name Funcd is arbitrary.
    Doub operator() (VecDoub_I &x)
    {
        return x[0]*x[0]+x[1]*x[1];
    }
    void df(VecDoub_I &x, VecDoub_O &deriv)     Name df is fixed.
    {
        deriv[0]=2.0*x[0];
        deriv[1]=2.0*x[1];
    }
};
```

To use `frprmn`, you need statements like the following:

```
Funcd funcd;
Frprmn<Funcd> frprmn(funcd);
VecDoub p = ...;
p=frprmn.minimize(p);          OK to overwrite initial guess.
```

The function value at the minimum is available as `frprmn.fret`. Note that the constructor takes an optional argument that specifies the function tolerance for the minimization.

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 (§10.8.1).

mins_ndim.h

```
template <class T>
struct Frprmn : Linemethod<T> {
Multidimensional minimization by the Fletcher-Reeves-Polak-Ribiere method.
    Int iter;
    Doub fret;                                  Value of the function at the minimum.
    using Linemethod<T>::func;                  Variables from a templated base class
    using Linemethod<T>::linmin;                    are not automatically inherited.
    using Linemethod<T>::p;
    using Linemethod<T>::xi;
    const Doub ftol;
    Frprmn(T &funcd, const Doub ftoll=3.0e-8) : Linemethod<T>(funcd),
        ftol(ftoll) {}
    Constructor arguments are funcd, the function or functor to be minimized, and an optional
    argument ftoll, the fractional tolerance in the function value such that failure to decrease
    by more than this amount on one iteration signals doneness.
    VecDoub minimize(VecDoub_I &pp)
    Given a starting point pp[0..n-1], performs the minimization on a function whose value
    and gradient are provided by a functor funcd (see text).
    {
        const Int ITMAX=200;
        const Doub EPS=1.0e-18;
        const Doub GTOL=1.0e-8;
        Here ITMAX is the maximum allowed number of iterations; EPS is a small number to
        rectify the special case of converging to exactly zero function value; and GTOL is the
        convergence criterion for the zero gradient test.
        Doub gg,dgg;
        Int n=pp.size();                        Initializations.
        p=pp;
        VecDoub g(n),h(n);
```

```
        xi.resize(n);
        Doub fp=func(p);
        func.df(p,xi);
        for (Int j=0;j<n;j++) {
            g[j] = -xi[j];
            xi[j]=h[j]=g[j];
        }
        for (Int its=0;its<ITMAX;its++) {      Loop over iterations.
            iter=its;
            fret=linmin();                     Next statement is one possible return:
            if (2.0*abs(fret-fp) <= ftol*(abs(fret)+abs(fp)+EPS))
                return p;
            fp=fret;
            func.df(p,xi);
            Doub test=0.0;                     Test for convergence on zero gradient.
            Doub den=MAX(abs(fp),1.0);
            for (Int j=0;j<n;j++) {
                Doub temp=abs(xi[j])*MAX(abs(p[j]),1.0)/den;
                if (temp > test) test=temp;
            }
            if (test < GTOL) return p;         The other possible return.
            dgg=gg=0.0;
            for (Int j=0;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
                return p;                          we are already done.
            Doub gam=dgg/gg;
            for (Int j=0;j<n;j++) {
                g[j] = -xi[j];
                xi[j]=h[j]=g[j]+gam*h[j];
            }
        }
        throw("Too many iterations in frprmn");
    }
};
```

10.8.1 Note on Line Minimization Using Derivatives

Kindly reread §10.6. We here want to do the same thing, but using derivative information in performing the line minimization. Simply replace all occurrences of `Linemethod` in `Frprmn` with `Dlinemethod`. The routine `Dlinemethod` is exactly the same as `Linemethod` except that `Brent` is replaced by `Dbrent` and `F1dim` by `Df1dim`:

mins_ndim.h

```
template <class T>
struct Dlinemethod {
```

Base class for line-minimization algorithms using derivative information. Provides the line-minimization routine `linmin`.

```
    VecDoub p;
    VecDoub xi;
    T &func;
    Int n;
    Dlinemethod(T &funcc) : func(funcc) {}
```

Constructor argument is the user-supplied function or functor to be minimized.

```
    Doub linmin()
```

Line-minimization routine. Given an `n`-dimensional point `p[0..n-1]` and an `n`-dimensional direction `xi[0..n-1]`, moves and resets `p` to where the function or functor `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 the value of func at the returned location p. All of this is
    actually accomplished by calling the routines bracket and minimize of Dbrent.
    {
        Doub ax,xx,xmin;
        n=p.size();
        Df1dim<T> df1dim(p,xi,func);
        ax=0.0;                                   Initial guess for brackets.
        xx=1.0;
        Dbrent dbrent;
        dbrent.bracket(ax,xx,df1dim);
        xmin=dbrent.minimize(df1dim);
        for (Int j=0;j<n;j++) {                   Construct the vector results to return.
            xi[j] *= xmin;
            p[j] += xi[j];
        }
        return dbrent.fmin;
    }
};
```

mins_ndim.h

```
template <class T>
struct Df1dim {
Must accompany linmin in Dlinemethod.
    const VecDoub &p;
    const VecDoub &xi;
    Int n;
    T &funcd;
    VecDoub xt;
    VecDoub dft;
    Df1dim(VecDoub_I &pp, VecDoub_I &xii, T &funcdd) : p(pp),
        xi(xii), n(pp.size()), funcd(funcdd), xt(n), dft(n) {}
    Constructor takes as inputs an n-dimensional point p[0..n-1] and an n-dimensional direc-
    tion xi[0..n-1] from linmin, as well as the functor funcd.
    Doub operator()(const Doub x)
    Functor returning value of the given function along a one-dimensional line.
    {
        for (Int j=0;j<n;j++)
            xt[j]=p[j]+x*xi[j];
        return funcd(xt);
    }
    Doub df(const Doub x)
    Returns the derivative along the line.
    {
        Doub df1=0.0;
        funcd.df(xt,dft);                 Dbrent always evaluates the derivative at the
        for (Int j=0;j<n;j++)                 same value as the function, so xt is un-
            df1 += dft[j]*xi[j];              changed.
        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. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), §8.7.