## Methods that require only first derivatives and store no matrices

Reference: Section 9.5.2

Sometimes problems are too big to allow $n^2$ storage space for the Hessian matrix.

Some alternatives:

- steepest descent
- nonlinear conjugate gradient
- limited memory Quasi-Newton
- truncated Newton

## Low-storage Method 1: Steepest Descent

### Back to that foggy mountain

If we walk in the direction of steepest descent until we stop going downhill, we clearly are guaranteed to get to a local minimizer.

The trouble is that the algorithm is terribly slow.

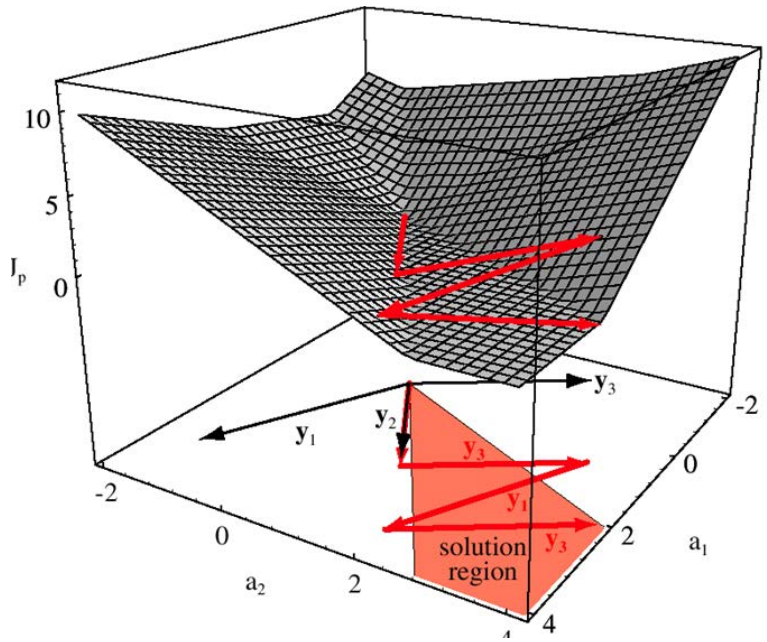
If we apply steepest descent to a quadratic function of $n$ variables, then after many steps, the algorithm takes alternate steps approximating two directions: those corresponding to the eigenvectors of the smallest and the largest eigenvalues of the Hessian matrix.

The convergence rate on quadratics is linear:

$$f(\mathbf{x}^{(k+1)}) - f(\mathbf{x}_{opt}) \leq \left(\frac{\kappa - 1}{\kappa + 1}\right)^2 (f(\mathbf{x}^{(k)}) - f(\mathbf{x}_{opt}))$$

where $\kappa$ is the ratio of the largest to the smallest eigenvalue of $\mathbf{H}$.

If steepest descent is applied to non-quadratic functions, using a good line search, then convergence is local and linear.



Advantages of steepest descent:

- No need to evaluate the 2nd derivative or to solve a linear system.
- Low storage: no matrices.

Disadvantages of steepest descent:

- Very slow.
- Very, very slow.

Use nonlinear conjugate gradients instead. Same advantages but better convergence.

# Low-storage Method 2: Nonlinear conjugate gradient methods

## Linear conjugate gradients

Reference: Section 28.2

The conjugate gradient method (Hestenes&Stiefel, 1952) is a method for solving linear systems of equations $\mathbf{Ax} = \mathbf{b}$ when $\mathbf{A}$ is symmetric and positive definite.

There are many ways to understand it, but for us, we think of it as minimizing the function

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Ax} - \mathbf{x}^T\mathbf{b}$$

which has gradient $\mathbf{g}(\mathbf{x}) = \mathbf{Ax} - \mathbf{b}$. So a minimizer of $f$ is a solution to our linear system.

We could use steepest descent, but we want something faster.

See the notes on the linear algorithm to understand how conjugate gradient combines the concepts of descent and conjugate directions.

## Summary of the linear conjugate gradient algorithm

Given $\mathbf{x}^{(0)}$, form $-\mathbf{g}(\mathbf{x}^{(0)}) = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)} = \mathbf{p}^{(0)}$.

For $k = 0, 1, \ldots$, until convergence,

Use a line search to determine $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{p}^{(k)}$.
Set $\mathbf{p}^{(k+1)} = -\mathbf{g}(\mathbf{x}^{(k+1)}) + \beta^{(k+1)}\mathbf{p}^{(k)}$.

## Linear conjugate gradients in more detail

Purpose: To solve $\mathbf{Ax} = \mathbf{b}$ when $\mathbf{A}$ is symmetric and positive definite. The only access to $\mathbf{A}$ is through a function call that returns the product of $\mathbf{A}$ with any given vector.

Given $\mathbf{x}^{(0)}$, form $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$ and $\mathbf{s}^{(0)} = -\mathbf{g}(\mathbf{x}^{(0)})$ (Note that $\mathbf{r}$ is the negative gradient of $f(\mathbf{x}) = 1/2\,\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{x}^T\mathbf{b}$, and I am using $\mathbf{s}$ for the search direction to avoid confusion later). For $k = 0, 1, \ldots$, until

convergence,

Let $\mathbf{z}^{(k)} = \mathbf{A}\mathbf{s}^{(k)}$.
Let the step length be $\alpha^{(k)} = (\mathbf{r}^{(k)T}\mathbf{s}^{(k)})/(\mathbf{s}^{(k)T}\mathbf{z}^{(k)})$.
Let $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{s}^{(k)}$.
Update the negative gradient $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha^{(k)}\mathbf{z}^{(k)}$.
Let $\beta^{(k+1)} = (\mathbf{r}^{(k+1)^T}\mathbf{r}^{(k+1)})/(\mathbf{r}^{(k)T}\mathbf{r}^{(k)})$.
Let the new search direction be $\mathbf{s}^{(k+1)} = \mathbf{r}^{(k+1)} + \beta^{(k+1)}\mathbf{s}^{(k)}$.

## The nonlinear conjugate gradient algorithm

Given $\mathbf{x}^{(0)}$, form $\mathbf{p}^{(0)} = -\mathbf{g}(\mathbf{x}^{(0)})$.

For $k = 0, 1, \ldots$, until convergence,

Use a line search to determine $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{p}^{(k)}$.
Set $\mathbf{p}^{(k+1)} = -\mathbf{g}(\mathbf{x}^{(k+1)}) + \beta^{(k+1)}\mathbf{p}^{(k)}$.

The parameter $\alpha$ is determined by a line search.

The parameter $\beta$ has many definitions that are equivalent for the linear problem but different when we minimize nonlinear functions:

$$\beta^{(k+1)} = \frac{\mathbf{g}^{(k+1)^T}\mathbf{g}^{(k+1)}}{\mathbf{g}^{(k)T}\mathbf{g}^{(k)}} \quad \text{Fletcher-Reeves}$$

$$\beta^{(k+1)} = \frac{\mathbf{y}^{(k)T}\mathbf{g}^{(k+1)}}{\mathbf{g}^{(k)T}\mathbf{g}^{(k)}} \quad \text{Polak-Ribiére}$$

$$\beta^{(k+1)} = \frac{\mathbf{y}^{(k)T}\mathbf{g}^{(k+1)}}{\mathbf{y}^{(k)T}\mathbf{p}^{(k)}} \quad \text{Hestenes-Stiefel}$$

Good theorems have been proven about convergence of Fletcher- Reeves, but Polak-Ribiére is generally better performing.

Note that this method stores no matrix. We only need to remember a few vectors at a time, so it can be used for problems in which there are thousands or millions of variables.

The convergence rate is linear, unless the function has special properties, but generally faster than steepest descent: for quadratics, the rate is

$$f(\mathbf{x}^{(k+1)}) - f(\mathbf{x}_{opt}) \leq \left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^2 (f(\mathbf{x}^{(k)}) - f(\mathbf{x}_{opt}))$$

where $\kappa$ is the ratio of the largest to the smallest eigenvalue of $\mathbf{H}$.

## An important property of conjugate gradients:

If we run cg on a quadratic function, then it generates the same iterates as the Huang family of Quasi-Newton algorithms.

## Low-storage Method 3: Limited-memory Quasi-Newton methods

Consider as an example the DFP formula

$$\mathbf{C}^{(k+1)} = \mathbf{C}^{(k)} - \frac{\mathbf{C}^{(k)}\mathbf{y}^{(k)}\mathbf{y}^{(k)T}\mathbf{C}^{(k)}}{\mathbf{y}^{(k)T}\mathbf{C}^{(k)}\mathbf{y}^{(k)}} + \frac{\mathbf{s}^{(k)}\mathbf{s}^{(k)T}}{\mathbf{y}^{(k)T}\mathbf{s}^{(k)}}$$

We've thought of this as a matrix stored in memory. Let's develop a different view.

How do we use this matrix? All we need to do is to multiply vectors, such as the gradient $\mathbf{g}^{(k)}$, by it.

Suppose we store $\mathbf{C}^{(0)}$. Then if we stored $\mathbf{C}^{(0)}\mathbf{y}^{(0)}$ and $\mathbf{s}^{(0)}$, plus the scalar values $\mathbf{y}^{(0)T}\mathbf{C}^{(0)}\mathbf{y}^{(0)}$ and $\mathbf{y}^{(0)T}\mathbf{s}^{(0)}$, we would be able to form products of $\mathbf{C}^{(1)}$ with any vector:

$$\mathbf{C}^{(1)}\mathbf{z} = \mathbf{C}^{(0)}\mathbf{z} - \frac{\mathbf{y}^{(0)T}\mathbf{C}^{(0)}\mathbf{z}}{\mathbf{y}^{(0)T}\mathbf{C}^{(0)}\mathbf{y}^{(0)}}\mathbf{C}^{(0)}\mathbf{y}^{(0)} - \frac{\mathbf{s}^{(0)T}\mathbf{z}}{\mathbf{y}^{(0)T}\mathbf{s}^{(0)}}\mathbf{s}^{(0)} .$$

We can play this game again for $\mathbf{C}^{(2)}$: if we store $(\mathbf{C}^{(0)}\mathbf{y}^{(0)}, \mathbf{s}^{(0)})$ and $(\mathbf{C}^{(1)}\mathbf{y}^{(1)}, \mathbf{s}^{(1)})$, with 4 scalars, then we can form products of $\mathbf{C}^{(2)}$ with any vector.

What have we accomplished? Recall that $\mathbf{C}^{(0)}$ is usually chosen to be the identity matrix, which requires no storage. Therefore, instead of taking $n^2$ storage locations for $\mathbf{C}^{(2)}$, we only need $4n + 4$!

The idea behind limited memory quasi-Newton algorithms is to continue this process $\ell$ steps, until we don't want to store any more vectors. Then, we start storing the new vectors in place of the oldest ones that we remember, always keeping the $\ell$ most recent updates.

For various more refined strategies, see reference 4 at the end of these notes.

## Low-storage Method 4: Truncated Newton methods

Again we return to the way the Hessian approximation is used.

If we have the Hessian matrix, how do we use it? Newton's method determines the search direction by solving the linear system

$$\mathbf{Hp} = -\mathbf{g}$$

We usually think of solving this by factoring $\mathbf{H}$ and then using forward- and back- substitution.

But if $\mathbf{H}$ is large, this might be too expensive, and we might choose to use an iterative method, like linear conjugate gradients, to solve the system.

If we do, How do we use the Hessian? All we need to do is to multiply a vector by it at each step of the algorithm.

Now Taylor series tells us that, if $\mathbf{v}$ is a vector of length $1$, then

$$\mathbf{g}(\mathbf{x} + h\mathbf{v}) = \mathbf{g}(\mathbf{x}) + h\mathbf{H}(\mathbf{x})\mathbf{v} + O(h^2)\,,$$

so

$$\mathbf{H}(\mathbf{x})\mathbf{v} = \frac{\mathbf{g}(\mathbf{x} + h\mathbf{v}) - \mathbf{g}(\mathbf{x})}{h} + O(h)\,.$$

Therefore, we can get an $O(h)$ approximation of the product of the Hessian with an arbitrary vector by taking a finite difference approximation to the change in the gradient in that direction.

This is akin to the finite-difference Newton method, but much neater, because we only evaluate the finite difference in directions in which we need it.

## The Truncated Newton strategy

So we'll compute an approximation to the Newton direction $\mathbf{p} = -\mathbf{H}^{-1}\mathbf{g}$ by solving the linear system $\mathbf{Hp} = -\mathbf{g}$ using the conjugate gradient method, computing approximate matrix-vector products by extra evaluations of the gradient.

We hope to obtain a superlinear convergence rate, so we need, by the theorem of Sec. 9.2.1, that

$$\frac{\|\text{our direction} - \text{Newton direction}\|}{\|\text{our direction}\|} \to 0$$

as the iteration number $\to \infty$.

We ensure this by

- taking enough iterations of conjugate gradient to get a small residual to the linear system.
- choosing $h$ in the approximation carefully, so that the matrix- vector products are accurate enough.

# Automated differentiation

## Automated differentiation

The most tedious and error-prone part of nonlinear optimization: writing code for derivatives.

An alternative: let the computer do it.

Automatic differentiation is an old idea:

- The forward (bottom-up) algorithm was proposed in the 1970s.
- The backward (top-down) algorithm was proposed in the 1980s.
- Reliable software (`Adifor`, etc., by Bischof, Griewank, ...) was developed in the 1990s.

## No-derivative methods

Reference: Section 9.5.3

These are methods of last resort, generally used when

- derivatives are not available.
- derivatives do not exist.

This is an area of very active research currently. We'll consider three classes of methods.

## No-derivative Method 1: Finite difference methods

We could do finite differences on the function to get an approximate gradient, but this is not usually a good idea, given that automatic differentiation methods exist.

## No-derivative Method 2: Simplex-based methods

(Not to be confused with the simplex method for linear programming.)

The most popular of these is the Nelder-Meade algorithm, and Matlab has an implementation of this.

Idea:

- Suppose we have evaluated the function at the vertices of a simplex. (In 2-dimensions, this is a triangle, in 3, it is a tetrahedron, etc.)
- We would like to move one vertex of this simplex, reflecting it around its current position, until we have enclosed the minimizer in the simplex.
- Then we would like to shrink the size of the simplex to hone in on the minimizer.

Simplex-based algorithms have rather elaborate rules for determining when to reflect and when to shrink, and no algorithm that behaves well in practice has a good convergence proof.

For that reason, it looks as if they will fade in popularity, being supplanted by pattern search methods.

## No-derivative Method 3: Pattern search methods

Idea:

- Suppose we are given an initial guess $x$ at the solution and a set of at least $n+1$ directions $v_i$, $i = 1, \ldots, N$, that form a positive basis for $\mathcal{R}^n$: this means that any vector can be expressed as a linear combination of these vectors, where the coefficients in the combination are positive numbers.
- At each step, we do a line search in each of the directions to obtain $f(\mathbf{x} + \alpha_i \mathbf{v}_i)$ and replace $\mathbf{x}$ by the point with the smallest function value.

This is a remarkably simple algorithm, but works well in practice and is provably convergent!

Another desirable property is that it is easy to parallelize, and this is crucial to making a no-derivative algorithm effective when $n$ is large.

## Final words

- We have discussed a variety of algorithms for solving optimization problems.
- For least squares problems, use a specialized algorithm, as discussed in Chapter 13.
- Another class of algorithms (randomized algorithms) is discussed in Chapter 17.
- To choose among algorithms, see Section 9.6.