Conjugate Gradient Notes

Dec 2018 – Stuart Brorson, s.brorson@northeastern.edu

Preliminaries

Gradient descent and conjugate gradient are algorithms used to solve a system of linear equations,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2N}x_N &= b_2 \\ &\vdots \\ a_{N1}x_1 + a_{N2}x_2 + a_{N3}x_3 + \cdots + a_{NN}x_N &= b_N \end{aligned} \tag{1a}$$

We usually abbreviate this linear system using matrix notation as

$$Ax = b \tag{1b}$$

where A is an NxN matrix and x and b are (column) vectors.

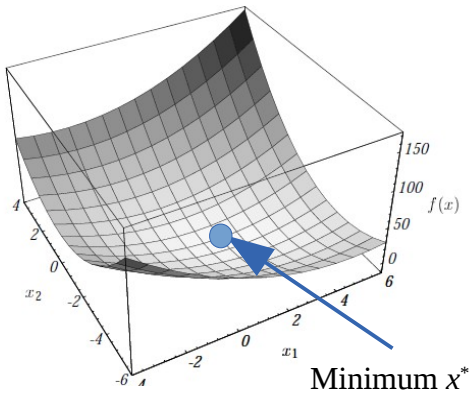
Both gradient descent and conjugate gradient are applicable in the case where the matrix A has the following properties:

- A is sparse.
- A is symmetric. That means the elements of A satisfy $a_{ij} = a_{ji}$, or equivalently $A = A^T$.
- A is positive definite. This means all eigenvalues of A are positive.

Both algorithms exploit the fact that solving the linear system Ax = b is equivalent to finding the x which minimizes the quadratic form,

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

Recall from class that one way you can visualize a matrix is to think about the (hyper-)surface mapped out by $f(x)$ as vector x is allowed to vary over its domain. When A is symmetric positive definite, the surface is an N-dimensional parabola. This is shown in the figure at right in the case where N = 2.



Minimum x^*

It turns out that the x satisfying eqs (1) corresponds to the minimum of the parabola. This can be seen by recalling the minimum of a function $f(x)$ occurs at the value of x where its gradient is zero, i.e.

$$\nabla f(x) = 0$$

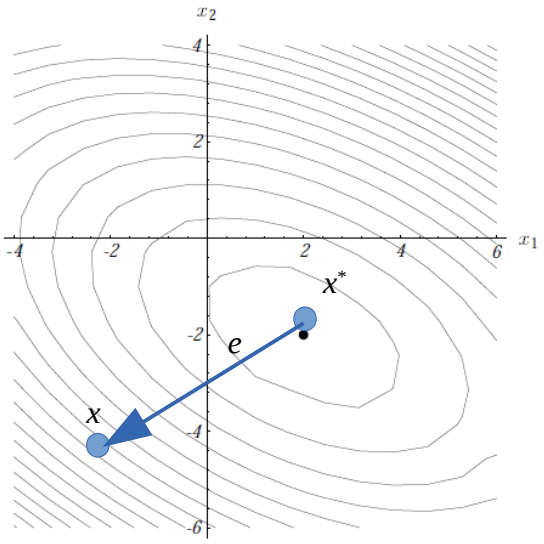
An easy calculation demonstrates that the gradient of (2) is

$$\nabla f(x) = Ax - b \tag{3}$$

and when the gradient is zero, we recover the linear system (1b).

Definitions

Levelsets: A levelset of a surface $f(x)$ is the set of all points x for which $f(x) = C$, where C is a constant. For the parabolic surfaces defined by eq (2), you can imagine cutting the parabola with a horizontal plane. The points lying on the intersection of the plane and the parabola typically form an ellipse. Those points are a levelset of the parabola. The levelsets of a typical parabola are shown in the figure at right.



Error: Imagine you are sitting at a point x away from the minimum x*. The error vector e is defined as

$$e = x - x^*$$

Note that the error vector e cannot be calculated unless you know x^*. Since the goal of solving eq (1) is finding x^*, the vector e is not knowable until the equation is solved.

Residual: Imagine you are sitting at point x. The residual vector r when sitting at point x is defined as

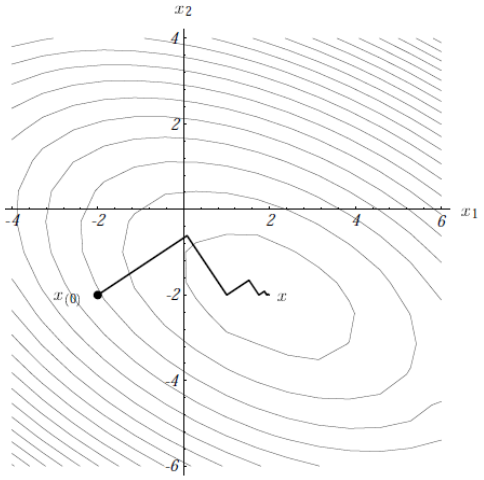
$$r = b - A x \tag{4}$$

Since you know all of x, A, and b, the residual may be calculated at any time. Note that the residual is the negative of the gradient of $f(x)$, that is,

$$r = -\nabla f(x) = b - A x$$

Gradient descent

Gradient descent is an iterative algorithm. The idea is to choose a random starting point x_0, then make a step towards the minimum point x^*. After you have stepped, you are at a new point which we will call x_1. Then take another step to point x_2, and repeat stepping until you think you are close enough to the minimum x^* that you can stop. This is illustrated in the figure at right which shows a series of steps converging on the minimum point of the parabola.



The question is, how to compute each step? Taking a step requires two pieces of information: a step direction and a step length. Then, each step taken in gradient descent is represented as

$$x_{n+1} = x_n + \alpha_n d_n \tag{5}$$

where x_n is the starting point, α_n is the step distance and d is the step direction.

Moving parts

The gradient descent algorithm uses the following moving parts to get information about how to step:

1. **Step direction.** Gradient descent computes the gradient of $f(x)$ at the point x to get a step direction. In particular, the gradient descent algorithm makes the very reasonable assumption

that the best direction in which to step is in the negative gradient direction:

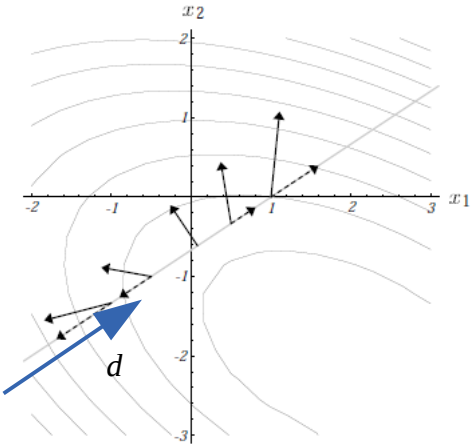
$$d_n = -\nabla f(x_n)$$

where d is the step direction vector. Gradient descent exploits the fact that d_n points in the "fastest downhill" direction of the parabola $f(x)$.

An important thing to note here is that the step direction d_n is also the residual r,

$$d_n = -\nabla f(x_n) = b - A x_n = r_n \qquad (6)$$

We'll use this fact shortly.



2. **Step length.** The optimum step length α corresponds to finding the lowest point along the line running in the direction d. The key observation to make is that $f(x)$ is minimized along the line d when d is tangent to a levelset. Equivalently, the minimum point is found when d is orthogonal to grad($f(x)$). These facts are illustrated in the figure on right.

   In class we derived this expression for the step distance:

$$\alpha_n = \frac{d_n^T d_n}{d_n^T A d_n} \qquad (7)$$

The algorithm

Equations (6), (7), and (5) form the gradient descent method. Here are all the pieces put together:

1. Assume we are at point x_n .
2. Get the step direction by computing the gradient at this point. Per (6), the gradient is simply the residual. Therefore, compute

$$d_n = r_n = b - A x_n$$

3. Get the step length

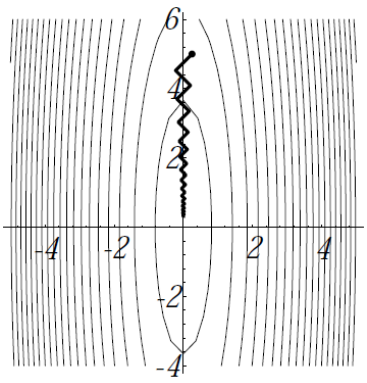
$$\alpha_n = \frac{d_n^T d_n}{d_n^T A d_n}$$

4. Take the step to the next point

$$x_{n+1} = x_n + \alpha_n d_n$$

5. Check for convergence. If converged, return. If not converged, loop back to 2.

Observations about gradient descent



Zig-zag behavior and convergence. A major flaw of gradient descent is that it tends to zig-zag when looking for the minimum point. This behavior results when the levelsets of $f(x)$ are non-circular.

Recall that the matrix condition number of A is a measure of how non-circular the levelsets of $f(x)$ are. Condition numbers around 1 correspond to nice, circular levelsets, while large condition numbers correspond to strongly elliptical levelsets. In the general case, your matrix A will not be well-conditioned. Therefore, gradient descent will slowly zig-zag its way to the minimum point. Accordingly, the method requires many iterations to achieve convergence, and the algorithm tends to be slow. This is a disadvantage, and means gradient descent is rarely used in practice as a way to solve linear systems.

A shortcut to compute r_i. Shewchuck [1] points out that the matrix-vector multiplication $A\,x$ in (6) may be replaced by an update rule for the residual,

$$r_{n+1} = r_n - \alpha_n A\, r_n \tag{8}$$

This works for n = 1, 2, 3..... and saves one matrix-vector multiplication at each step. (You must still compute $A\,r$, but not $A\,x$.) For $n = 0$ the expression (6) must be used. This trick is exploited in the conjugate gradient algorithm.

Conjugate gradient

Conjugate gradient is also an iterative algorithm, similar to gradient descent. However, conjugate gradient fixes the zig-zag behavior evidenced by gradient descent, and therefore converges quickly. Unfortunately, conjugate gradient relies on several subtle ideas; many people – including me – find it hard to understand when they first encounter the algorithm. My goal here is to present the different ideas and then show how the fit together to produce the conjugate gradient algorithm.

Inner product space of A

To start, how to fix the zig-zag behavior of gradient descent? Recall that when the levelsets of $f(x)$ are circular, gradient descent finds the minimum point in one step. This is because the gradient points directly to the center of the circular levelset. Knowing this, what if we could work in a space where the levelsets were circular?

It turns out that if you replace the ordinary definition of a dot product

$$u \cdot v = \langle u | v \rangle$$

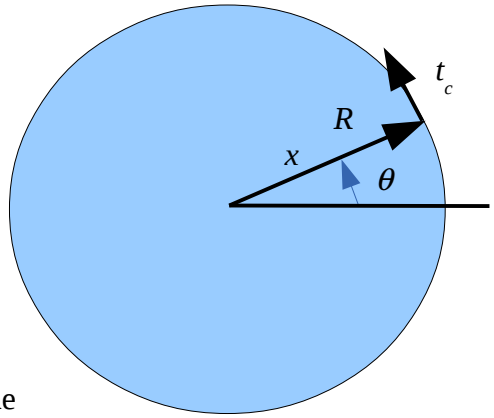
with the following dot product

$$u \cdot v = \langle u | A | v \rangle$$

then the space you work in is one in which the levelsets of $f(x) = \frac{1}{2} x^T A\, x - x^T b$ appear circular from the standpoint of deciding search directions – which is what taking the gradient is doing in gradient descent. One consequence of this is that in the A-inner product space the dot product of the error vector e and a tangent vector to the level set t are perpendicular. Here is the argument:

Circle. Consider the circle defined by the equation

$$x^T x = R^2 \qquad (9)$$



where x is the error vector (radius vector) shown in the drawing. R is the radius of the circle. The set of all x satisfying (9) defines the circumference of the circle.

Any radius vector x can be written as

$$x = R\begin{pmatrix}\cos(\theta)\\ \sin(\theta)\end{pmatrix}$$

Now consider the tangent vector to the circle, t_x. An expression for the tangent vector can be obtained by differentiating x,

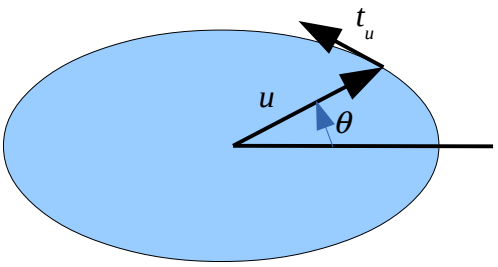
$$t_x = \frac{dx}{d\theta}$$

$$= R\begin{pmatrix}-\sin(\theta)\\ \cos(\theta)\end{pmatrix}$$

By inspection of the figure x and t_x are orthogonal on a circle, and this may be verified by computing the usual dot product $x^T t_x$ using the definitions of x and t_x above.

Ellipse. Now consider the ellipse created from the unit circle upon multiplying by the matrix B

$$u = Bx$$



This ellipse has a tangent vector t_u, given by

$$t_u = \frac{du}{d\theta}$$

$$= B\frac{dx}{d\theta}$$

Now consider the relationship between A – the original matrix which comes from the levelset point of view, and B – a matrix which transforms a circle to an ellipse via matrix-vector multiplication. Recall from class that

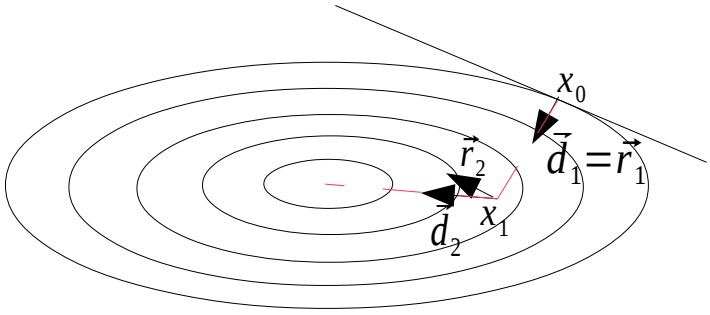
$$A = (B^{-1})^T (B^{-1})$$

What is the relationship between the vector from the ellipse center u and the tangent vector t_u? if we compute $\langle u | t_u \rangle_A$, we get

$$\begin{aligned}\langle u | t_u \rangle_A &= u^T A t_u \\ &= x^T B^T A B t_x \\ &= x^T B^T (B^T)^{-1} (B^{-1}) B t_x \\ &= x^T t_x \\ &= 0\end{aligned}$$

The conclusion is that u and t_u are A-orthogonal.

Next, just like in gradient descent, conjugate gradient will find a step length which will locate the minimum point of the ellipse along the direction of search. This point occurs where the search line is tangent to a levelset. Therefore, when computing a new next step direction at this point, we exploit the A-inner product instead of the usual inner product to compute *d*. The idea is that the step vector *d* computed in this way will point directly to the center of the ellipse.



Krylov subspaces

In gradient descent, each new step direction is found by computing the residual (4),

$$d_n = r_n = b - A\,x_n$$

then a new position is found

$$\begin{aligned} x_{n+1} &= x_n + \alpha_n d_n \\ &= x_n + \alpha_n (b - A\,x_n) \end{aligned}$$

Upon stepping again, we get

$$\begin{aligned} x_{n+2} &= x_{n+1} + \alpha_{n+1} d_{n+1} \\ &= x_{n+1} + \alpha_{n+1} (b - A\,x_{n+1}) \\ &= x_{n+1} + \alpha_{n+1} (b - A\,(x_n + \alpha_n (b - A\,x_n))) \end{aligned}$$

The thing to note here is that each iteration multiplies the last iteration by the matrix A. Therefore, the n^{th} iteration will include a term containing the matrix power A^n . Why is this important? It turns out that if you start with an arbitrary vector *u* and a full rank matrix *A* of size NxN, then you can form a linearly-independent basis set of vectors which span the space of *A* by forming powers of *A*. The following table summarizes this fact

Basis set	**Rank**
$\{u\}$	1
$\{u, A\,u\}$	2
$\{u, A\,u, A^2 u\}$	3
$\{u, A\,u, A^2 u, \ldots, A^{N-1} u\}$	*N*

Conjugate gradient exploits this fact as follows:

1. In the first iteration, we start with the residual as the search direction. The algorithm has a single search direction to work in. Conjugate gradient finds the minimum point along this search line.
2. In the second iteration, the new residual contains the first power of *A*. Our search space has two dimensions. Conjugate gradient finds the minimum point in this two dimensional search space in one step.
3. In the third iteration, the new residual contains a term in A^2. Therefore, the search space has three dimensions, and conjugate gradient finds the minimum in this space in one step.
4. In the M^{th} iteration, conjugate gradient works in an M-dimensional space, and finds the minimum in one step.
5. In principle, conjugate gradient will find the solution to the N dimensional problem $Ax = b$ in N steps. In practice, finding the solution can take more steps due to round-off errors.

Vector spaces generated by successive matrix powers are called "Krylov spaces", so you will often hear conjugate gradient referred to as a "Krylov method". There is a large family of so-called Krylov method solvers, and conjugate gradient is the simplest.

Mutually A-orthogonal basis vectors

Each iteration of conjugate gradient creates a vector pointing in a new dimension of the search space. However, these vectors point in arbitrary directions, i.e. they are not orthogonal in any sense, A- or otherwise.

Therefore, we next we consider how to span the space of *A* using a basis set which is mutually orthogonal in the A-inner product sense. Then, assuming the basis vectors are d_n , we can express the solution to the $A\,x = b$ problem, *x** as

$$x^* = \sum_{n=1}^{N} \alpha_n d_n$$

where *N* is the dimension of the space we are working in, i.e. the size of matrix A. The position of the solution depends upon generating a set of *N* direction vectors d_n , and then computing the distances α_n . When you iterate the conjugate gradient method, these vectors are generated by the process, one new vector with each step of the iteration.

How does conjugate gradient get a set of basis vectors d_n ? Recall the Gram-Schmidt procedure for finding an orthonormal set of basis vectors from an input of *N* arbitrary vectors. In the case of conjugate gradient, we use the residual at each step, r_n , as the new input vector to the Gram-Schmidt process. The new residual has terms arising from a matrix power, so it has a component pointing into a new direction. Then, with each iteration we use the new r_n to create a new d_n by shaving off the components of r_n which point into old directions. (Old in the A-orthogonal sense.) That is, the new d_n is constructed to be A-orthogonal to the previous d_n vectors. The construction method is the usual Gram-Schmidt process, but using the A-inner product:

$$d_n = r_n - \sum_{j=1}^{N-1} \frac{d_j^T A r_n}{d_j^T A d_j} d_j$$

When iterating the conjugate gradient algorithm, we don't need to perform the entire sum with each iteration. Rather, previous values are preserved during the iteration process and so we only need to update d_n with one calculation at each iteration. The update rule is presented below.

Moving parts

The following quantities are the objects computed by conjugate gradient when solving $Ax = b$.

Step direction: Stepping in conjugate gradient looks similar to stepping in gradient descent, except that the step vector d_n is the result of an extended Gram-Schmidt process. This process is obscured in the computations performed at each iteration, but the important pieces are:

- A scalar factor beta which ensures that d_{n+1} will be A-orthogonal to d_n :

$$\beta_{n+1} = -\frac{r_{n+1}^T r_{n+1}}{r_n^T r_n}$$

- The step direction itself:

$$d_{n+1} = r_{n+1} + \beta_{n+1} d_n$$

The idea is that the step vector *d* computed in this way will point directly to the center of the ellipse upon each iteration of the method. Note that the step direction *d* and the residual vector *r* generally point in different directions.

Step length: The step length looks similar to the step length used in gradient descent, except that in conjugate gradient the step direction d_n is not the same as the residual r_n . Therefore, in conjugate gradient, the formula for the step length computes something different from gradient descent. The formula is

$$\alpha_n = \frac{r_n{}^T r_n}{d_n{}^T A\, d_n}$$

Note that the formula involves both d_n and r_n .

Residual update: This is the same as used in gradient descent (8), except the search direction d_n is used. Keep in mind that the residual and the step direction are no longer the same quantity.

$$r_{n+1} = r_n - \alpha_n A\, d_n$$

The algorithm

Here are the steps in the conjugate gradient algorithm presented in order.

1. Start at initial point x_0 .
2. Compute the residual using the usual expression. Note that the residual is the same as the gradient at this point, just as in gradient descent.

$$r_0 = b - A \cdot x_0$$

3. Generate the initial step direction vector.

$$d_0 = r_0$$

4. Compute the desired step length.

$$\alpha_n = \frac{r_n^T r_n}{d_n^T A d_n}$$

5. Step to the next point.

$$x_{n+1} = x_n + \alpha_n d_n$$

6. Compute the residual at this point. Note that we replace the expression *Ax* with the update *Ad*, similar to (8) above. However, note that while (8) used the update *Ar*, in conjugate gradient we update w.r.t. the step direction vector which is formed using the A-inner product.

$$r_{n+1} = r_n - \alpha_n A d_n$$

7. Check for convergence and exit if the residual is less than some tolerance. The return quantity is x_{n+1} .

$$\|r_{n+1}\| < tolerance$$

8. Compute new beta. This quantity is related to the underlying Gram-Schmidt process and ensures that successive d vectors are A-orthogonal to each other.

$$\beta_{n+1} = -\frac{r_{n+1}^T r_{n+1}}{r_n^T r_n}$$

9. Compute a new step direction vector.

$$d_{n+1} = r_{n+1} + \beta_n d_n$$

10. Loop back to step 4 and continue iterating.

Observations about conjugate gradient

Code optimization. As presented above, the product *Ad* is formed twice, in step 4 and step 6. Since matrix-vector multiplication is $O(N^2)$, a good implementation will create a variable $z = Ad$ once, then use it in steps 4 and 6 rather than performing the same expensive multiplication twice.

Matlab. Matlab provides the basic conjugate gradient via the function pcg(). However, this is not the only Krylov method provided by Matlab. Matlab also provides more advanced methods such as bicg, gmres, minres, qmr and so on. These functions are actually wrappers around Lapack functions. Further information on the universe of iterative solvers is available at the Netlib website [2].

References

[1] "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain", Jonathan Richard Shewchuk. http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf

[2] "Iterative Methods", in "Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition", R. Barrett and M. Berry and T. F. Chan and J. Demmel and J. Donato and J. Dongarra and V. Eijkhout and R. Pozo and C. Romine and H. Van der Vorst. http://www.netlib.org/linalg/html_templates/node9.html