A BRIEF INTRODUCTION TO THE CONJUGATE GRADIENT METHOD

Runar Heggelien Refsnæs, Fall 2009

Introduction

The conjugate gradient(CG) method is one of the most popular and well known iterative techniques for solving sparse symmetric positive definite(SPD) systems of linear equations. It was originally developed as a direct method, but became popular for its properties as an iterative method, especially following the development of sophisticated preconditioning techniques.

The intention behind this small note is to present the motivation behind the CG-method, followed by some details regarding the algorithm and its implementation. It is important to note that the mathematical derivation of the method is intentionally incomplete and missing important details. This is done in order to keep the note small and focus on the more intuitive characteristics of the CG-method, making it more suitable for a target audience with a limited mathematical background.

The Quadratic Test Function

A natural starting point in deriving the conjugate gradient method is by looking at the minimization of *the quadratic test function*

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

with b, $x \in \mathbb{R}^n$ and $A \in \mathbb{R}^{nxn}$, where A is assumed to be SPD.
Note: A symmetric matrix A is SPD(symmetric positive definite) if $x^T Ax > 0$ $\forall x \in \Omega$, or equivalently if all the egeinvalues of A are positive.

The minimizer x^* of the function ϕ is given as the point where the gradient of the function is equal to zero. Direct calculation gives

$$\nabla\phi(x^*) = Ax^* - b = 0$$

or

$$Ax^* = b$$

This shows that the same x^* that minimizes $\phi(x)$ also serves as the solution to the linear system of equations $Ax = b$, motivating us to find a method capable of minimizing ϕ instead of looking at a direct solver of the algebraic system.

The uniqueness of our solution is guaranteed by the SPD condition. To see this, consider what you learned about unqiue minimas and maximas in high school(vgs). A minimum point is unique and global if the double derivative is larger than zero for all points of the domain. For multiple dimensions this is the same as requiring that *the hessian(or hess matrix)* $H = \nabla^2 > 0$.

For the quadratic test function this means that $\nabla^2\phi(x) = A$. But since A is SPD, $x^TAx > 0$, and hence our solution x^* is unique. This uniqueness can also be seen intuitively by considering the strict convexity of the function.

Line Search Methods

In the previous section we concluded that minimizing the quadratic test function is equivalent to solving $Ax = b$. We now need a strategy for solving our optimization problem.

The line search methods are a large family of iterative optimization methods where the the iteration is given by

$$x_{k+1} = x_k + \alpha_k p_k$$

The idea is to choose an initial position x_0, and for each step walk along a direction(a line) so that $\phi(x_{k+1}) < \phi(x_k)$. The different methods have various strategies for how they choose the *search direction* p_k, and the *step length* α_k.

The *steepest descent method* is perhaps the most intuitive and basic line search method. We remember that the gradient of a function is a vector giving the direction along which the function increases the most. The method of steepest descent is based on the strategy that in any given point x, the search direction given by the negative gradient of the function $\phi(x)$ is the direction of steepest descent. In other words, the negative gradient direction is the locally optimal search direction. We have already found the gradient of $\phi(x)$ to be $Ax - b$. We also call this *the residual* r of the system.

We now have our direction, but we still need to know how far to walk along it. The natural choice is obviously to walk until we no longer descend, and an expression for the optimal step length α_k is easily found to be

$$\alpha_k = \frac{\nabla\phi(x_k)^T\nabla\phi(x_k)}{\nabla\phi(x_k)^TA\nabla\phi(x_k)} = \frac{{r_k}^T r_k}{{r_k}^T A r_k}$$

by inserting the expression for the next step $x_{k+1} = x_k - \alpha\nabla\phi(x_k)$ into the quadratic test function and minimizing with respect to α. We repeat this for every step, taking the gradient of ϕ in the next point x_{k+1} and by finding a new step length. It is obvious that this leads to search directions that are orthogonal to each other, as can be seen in figure [1]

This means that the method zigzag towards the solution. While it can be shown that the steepest descent method always converge for a SPD problem like ours, it seems obvious that its zigzag behaviour is not the optimal

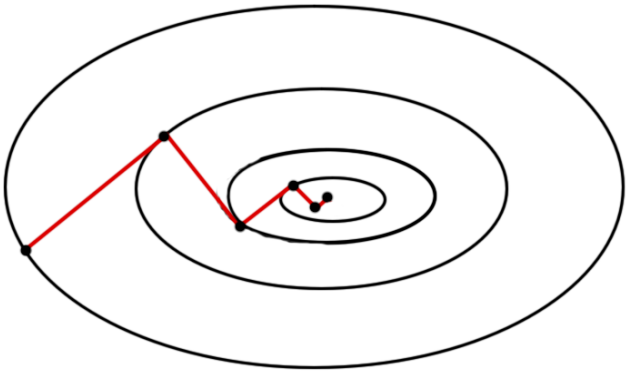

Figure 1: The contour plot of a function, with the steps of the steepest descent method in red

and fastest path towards the minimum. The problem is simply that each succesive step is not different enough from the others. We need a method that somehow uses the information from the previous steps in order to avoid running back and forth across the valley of our contour plots.

The Conjugate Gradient Method

We now turn our eye to *the A-conjugate direction methods*. A-conjugacy means that that a set of nonzero vectors $\{p_0, p_1, ..., p_{n-1}\}$ are *conjugate* with respect to the SPD matrix A. That is

$$p_i{}^T A p_j \qquad \forall i \neq j$$

A set of n such vectors are linearly independent and hence span the whole space $\mathbb{R}^n$. The reason why such A-conjugate sets are important is that we can minimize our quadratic function ϕ in n steps by successively minimizing it along each of the directions. Since the set of A-conjugate vectors acts as a basis for $\mathbb{R}^n$, we can express the difference between the exact solution $x^*$and our first guess x_0 as a linear combination of the conjugate vectors:

$$x^* - x_0 = \sigma_0 p_0 + \sigma_1 p_1 + ... + \sigma_{n-1} p_{n-1}$$

By utilizing our conjugacy property, we find that the coefficients σ_k are the same as the step lengths α_k that minimize the quadratic function ϕ along $x_k + \alpha_k p_k$, and hence:

$$x^* = x_0 + \alpha_0 p_0 + \alpha_1 p_1 + ... + \alpha_{n-1} p_{n-1}$$

You can think of this as gradually building the solution along the dimensions of our solution space. In fact, for diagonal matrices, the conjugate search vectors coincide with the coordinate axes. In each step k, x_k is the exact solution x^* *projected* into the solution space spanned by the k vectors.

All of this sounds great, but so far we have just assumed that the set of A-conjugate search directions exists. In practice we need a way to create it. There are several ways to choose such a set. *The eigenvectors* of A form a A-conjugate set, but finding the eigenvectors is a task requiring a lot of computations, so we better find another strategy. A second alternative is to modify the usual *Gram-Schmidt orthogonalization process*. This is also not optimal, as it requires storing all the directions.

It turns out there is a conjugate direction method with the very nice property that each new conjugate vector p_k can be computed by using only the previous vector p_{k-1}. Without knowing the other previous vectors the new vector is still automatically conjugate to the others. The method works its magic by chosing each new direction p_k as a linear combination of the negative residual $-r_k$ and the previous search vector p_{k-1}. We remember that for the quadratic function the negative residual is equal to the steepest descent or negative gradient direction. Hence the name of the method: *the conjugate gradient method*.

The formula for the new step becomes

$$p_k = -r_k + \beta_k p_{k-1}$$

where β_k is found by imposing the condition that ${p_{k-1}}^T A p_k$ and is given as

$$\beta_k = \frac{{r_k}^T A p_{k-1}}{{p_{k-1}}^T A p_{k-1}} = \frac{{r_k}^T r_k}{{r_{k-1}}^T r_{k-1}}$$

A comparison of the conjugate gradient method and the steepest descent method can be seen in figure [2]

Algorithm and Implementation

We are finally ready to write up the algorithm for the conjugate gradient method. While we have not covered all the details of its derivation, we should still be able to recognize most of the steps of the iteration, and what they do.

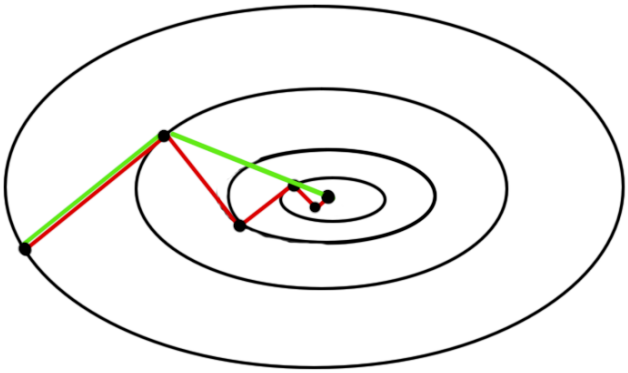

Figure 2: The contour plot of a function, with the steps of the steepest descent method in red and of the conjugate gradient method in green

The conjugate gradient algorithm
Compute $r_0 = Ax_0 - b, p_0 = -r_0$
For $k = 0, 1, 2, ..$ until convergence

$\alpha_k = \frac{{r_k}^T r_k}{{p_k}^T A p_k}$
$x_{k+1} = x_k + \alpha_k p_k$
$r_{k+1} = r_k + \alpha_k A p_k$
$\beta_k = \frac{{r_{k+1}}^T r_{k+1}}{{r_k}^T r_k}$
$p_{k+1} = -r_{k+1} + \beta_k p_k$

End

The first step is to find the initial residual which is the same as the first gradient search direction. If the initial solution x_0 is zero, then r_0 and p_0 simply becomes b. Inside the for-loop we recognize the first line as the calculation of the step length. In the second line we update our solution by adding our step. Then in line three we update our residual, and line 4 and 5 gives us the new search direction.

For an actual implementation of the algorithm above, there are several things to consider. First we notice that both $r_k^T r_k$ and Ap_k are each used twice for every iteration. Also $r_{k+1}^T r_{k+1}$ is simply ${r_k}^T r_k$ in the next iteration. There is no sense in doing the same calculations twice, so one should store the results the first time they are calculated and reuse them in the calculations that follow.

Another thing to consider is the matrix-vector calculation Ap_k. When doing

a numerical analysis of partial differential equations, our matrix typically comes from a finite difference discretization or from the stiffness assembly of the finite element method. This means that A can be very large, but also very sparse(few non-zero elements) and often with a clear structure(at least for the finite difference methods) such as the tri-diagonal banded matrix from the 2D poisson problem.

A trick is to avoid construction and storing of the actual matrix, and instead do the equivalent calculations explicitly for each node or element in the problem. This way we can save both valuable memory and floating point operations. Using the specific knowledge of the problem you are trying to solve can often be a key factor in achieving good performance, and in general, numerical solvers that do this will outperform more general solvers.

Finally the 'until convergence' condition in our for-loop is not very precise. As mentioned, CG will theoretically reach the exact solution in no more than n steps where n is the size of the matrix A. However, there is no point in doing all those iterations if our solution is already within a tolerable margin of error. Furthermore, when doing numerical analysis of PDEs it is important to remember that the exact solution given by the CG-method is only the exact solution of our discretized system, not the exact solution of our PDE. This means that there is no point in running the CG-method towards an error of value zero(tehcnically this is not even possible, due to the round-off error of floating point numbers), instead one should try to balance the discretization error and the error in the CG-solver.

Preconditioning

The CG-method is an excellent algorithm, being both extremely fast but also very easy to implement. However, it turns out through numerical experiments that alot of times the pure CG-method simply does not converge as fast as we would like. The reason for this slow convergence is that the system is *ill-conditioned*. An ill-conditioned system is a system with a high *condition number* κ. The condition number for a system like ours can be expressed as $\kappa = \frac{\lambda_{max}}{\lambda_{min}}$. It can be shown that for the conjugate gradient method, the number of iterations required to reach convergence is proportional to the condition number: $N_{it} \sim \mathbb{O}(\kappa)$.

In the introduction of this document, there was briefly mentioned that the CG-method owes much of its popularity to good preconditioning techniques. Preconditioning is simply a way of manipulating the original system in a way that improves its condition number, and hence its rate of convergence. The preconditioner itself is nothing more than a matrix P such that $P^{-1}A$

has a better condition number. The obvious best choice is $P = A$ wich actually solves the problem, but of course if we had A^{-1} we wouldn't really be interested in preconditioning of the CG-method in the first place. In the opposite end we find the cheapest and least effective preconditioner $P = I$ which does absolutely nothing. In practice one would try to find something in between.

There is much to be said about preconditioning, and finding a good way of doing preconditioning can be a very difficult task, but can likewise lead to very impressive convergence results. A lot of the time the only real way of deciding what works best is by doing experiments.

Summary

We started our discussion of the conjugate gradient method by noticing that a linear system of equations $Ax = b$ could be written as a minimization problem of the quadratic test function $\phi(x) = \frac{1}{2}x^T Ax - x^T b$. We then introduced the line search methods as an iterative approach, giving each new step as $x_{k+1} = x_k + \alpha_k p_k$. Our first atempt at using line search was the steepest descent method, but this gave us slow convergence bacause of its zigzag movement. Changing our focus to the A-conjugate direction methods, we found that by using information from the previous steps, we could get the exact solution in n or less steps. In our desire to build the set of A-conjugate directions as cheaply as possible, we finally ended up with the conjugate gradient method. Still hungry for better performance we then introduced preconditioning as a way to get even faster convergence.