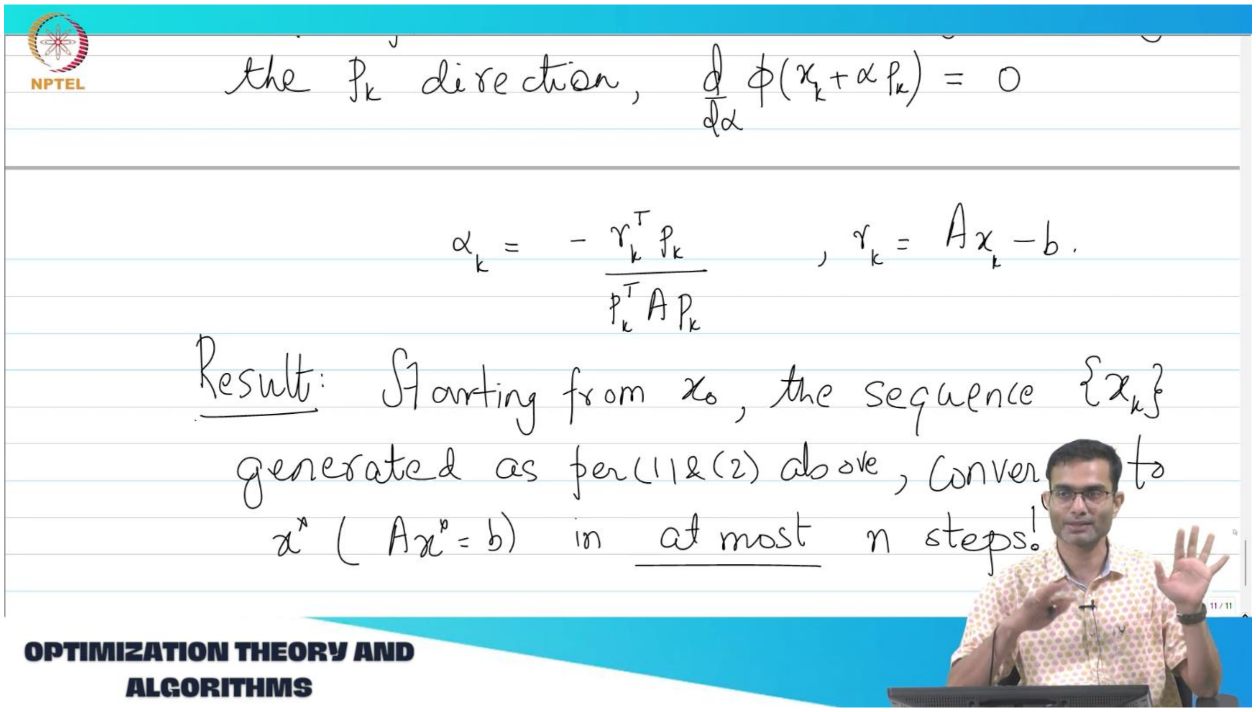
Course Name: Optimization Theory and Algorithms
Professor Name: Dr. Uday K. Khankhoje
Department Name: Electrical Engineering
Institute Name: Indian Institute of Technology Madras
Week - 05
Lecture - 36

Conjugate Directions Method - Introduction and Proof

Now, it turns out that if you study the field of this method, before the conjugate gradient method came into existence, its ancestor was a similarly named method called the conjugate directions method. These p vectors look like conjugate directions, right? So, just starting on the basis of this, the method that people invented was the conjugate direction method, and we will see it ran into some computational problems. What solved it was the conjugate gradient method. So, we need to first understand the conjugate directions method.

Let us call this method CDM, and note that it is the ancestor of the conjugate gradient method. Now, what we have, as I mentioned, is $Ax = b$, and I do not want to solve the system of equations directly. If I do not want to solve something directly, what is my option, what is my alternative? Something that is not direct is called what? Well, obviously indirect, but something else... If I do not get the solution in one shot, what do I call that method? Iterative method, right. Was our gradient descent an iterative method? Yes, because I took discrete steps, and each step got better, right. So, this CDM is also an iterative method.

So, it is iterative, and guess what? I can write the iteration the same way that I wrote the iteration for a line search method. I want this, and the very interesting thing is that this p_k, I am not going to insist on it being a descent direction because then I am back to a gradient descent.



NPTEL

the p_k direction, $\frac{d}{d\alpha}\phi(x_k+\alpha p_k) = 0$

$$\alpha_k = -\frac{r_k^T p_k}{p_k^T A p_k}, \quad r_k = Ax_k - b.$$

Result: Starting from x_0, the sequence $\{x_k\}$ generated as per (1) & (2) above, conver to x^* ($Ax^* = b$) in $\underline{\text{at most}}$ n steps!

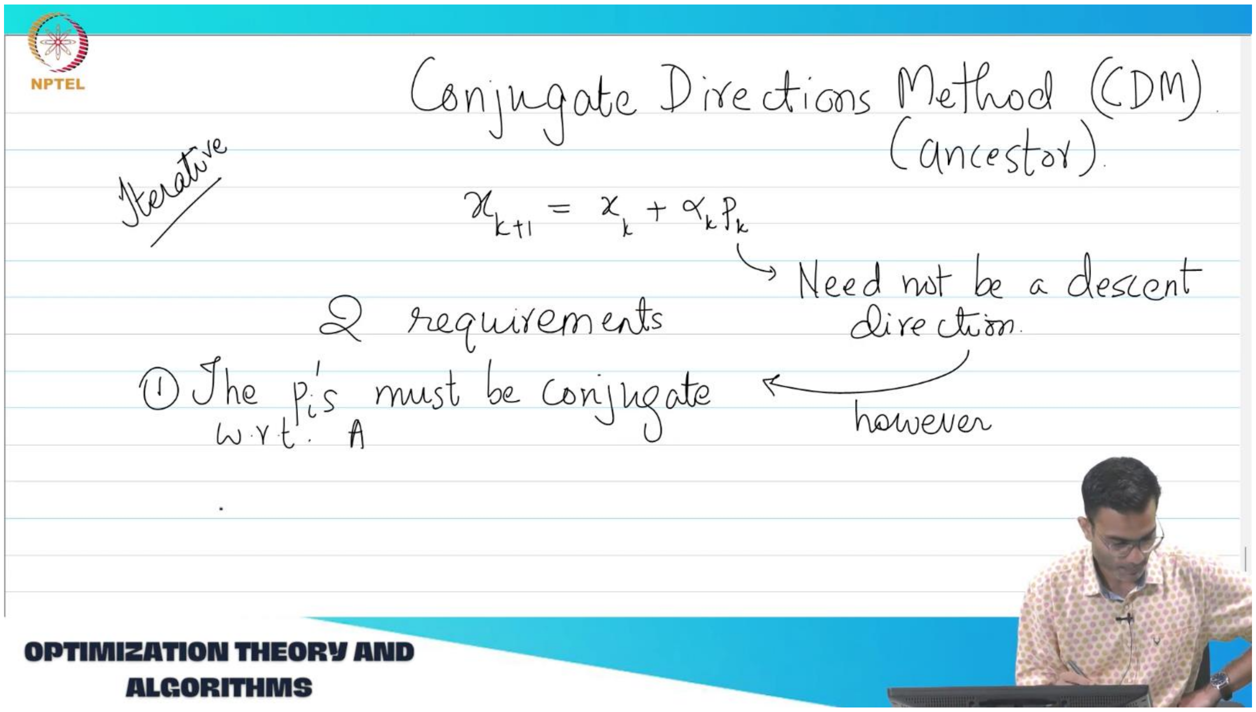
11 / 11

OPTIMIZATION THEORY AND ALGORITHMS

So, I am going to say that it need not be a descent direction. But just as we know, there is no free lunch, there are different constraints that I need to impose on this p. Let us put a "however." No points for guessing the first requirement—what is it? The p's that I choose should be conjugate with respect to A, meaning p_i's must be conjugate with respect to A.

The second step or requirement, rather, can also be guessed by you with a little bit of hints. When we did this in the tutorial, when we have a quadratic cost function, was it possible to find the exact step length? Was there a closed-form expression for the step length? Yes.

Is my cost function here quadratic? It is. So, does it make sense for me at all to do line search? There is no sense, right? I am wasting resources doing a line search when I know what the exact step length should be. So, the second requirement is that all the alphas are exact step lengths, exact minimizers of the cost function along which direction? If I am at the k-th step and I want to reach x_{k+1}, what direction am I going along? p_k, right? If I am at k and I want to go to $k+1$, I have to go in the direction p_k and a length α_k, and because my cost function is quadratic, I can use simple calculus to find the exact step length required to reach there. So, I can get a closed-form expression for α_k, and it is common sense to use the exact step when exact is available, right?



Conjugate Directions Method (CDM)

(ancestor)

Iterative

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

Need not be a descent direction.

however

2 requirements

① The p_i's must be conjugate w.r.t. A

OPTIMIZATION THEORY AND ALGORITHMS

So, that is the simple requirement—well, you can call it a requirement, but it is more like common sense. Obviously, you should use the exact step length when exact is available.

The step length is an exact minimizer of $\phi(x)$ along the direction p_k. In other words, again this is a calculus review. What should I write over here? I am at the k-th step. So, what is my variable? p_k, supposing it has been given to me. p_k has come from step 1 or requirement 1, right?

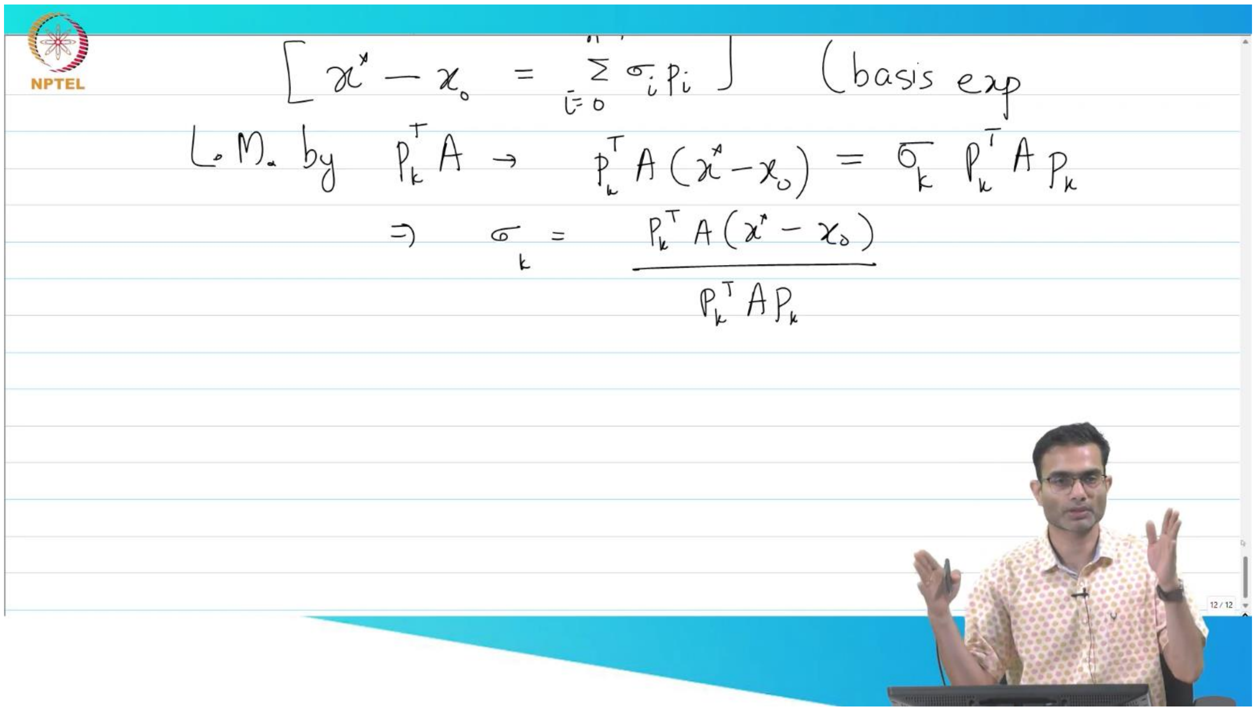
So, $\alpha_k p_k$ and I am going to set this equal to 0. The α_k which solves this is going to be given the name α_k, that is how we do it. In fact, this α_k has a very nice closed-form expression as was there in the tutorial, and I am going to write it over here. Remember r_k was nothing but $r_k = Ax_k - b$. So, I started with this whole business of conjugacy, and based on that, I said, "Okay, let me try to discover an iterative method." So, what are the nuts and bolts I started putting together? I said, "Okay, give me the directions in which I walk, they will be the p's." Alright, and since I am walking along the p's, I have the exact minimizer, so I get the alphas.

Now, what is really surprising, and it is not obvious looking at this, is that with these two simple requirements, here is what happens: you are guaranteed convergence of the problem in n steps. That was not possible for you in gradient descent. In gradient descent, depending on the condition number of A, you took more than the size of the matrix, that many steps. But here, this is a guarantee over here. So, let us write this down. Starting from x_0, we have the sequence x_k generated as per 1 and 2 above, which converges to the stationary point in at most n steps. It is so simple looking, so elegant, and all I had to do was to start by introducing just one new generalization of the idea of orthogonality into conjugacy. Any questions on the statement? The statement is clear.

If you generate step vectors like this, I have not told you how to generate the p's. That's the hint—the devil lies in those details, right? And you get the, if I give you the p's, for example, do you think α_k is complicated or difficult to compute? Look at the expression for α_k. It is just a matrix-vector product, right? It is going to take no time in MATLAB, for example, to evaluate this. And you are done in n steps. In fact, as I said, the devil lies in the details—that is where the CDM got stuck in terms of implementation.

The cost for generating the p's became very high, but we will get that. Now, obviously, I have written an extremely strong statement over here, and we would want to see a proof, right? Why is it that it converges in at most n steps? That means, what? It could also be fewer. So, let us look at the proof. Is everyone with me? So, let us say I am at some in-between k-th step, okay?

So, we are at step k. Obviously, I am going to assume k is less than n, that is a common-sense assumption. Assume $k \leq n$. Now, if I have reached k, how many—well, nothing to do with k, but if I have n as the size of the matrix A, how many conjugate, how many linearly independent vectors are possible in n dimensions? n, and no more, right?



$$\left[x^* - x_0 = \sum_{i=0} \sigma_i p_i \right] \quad (\text{basis exp}$$

$$\text{L.M. by } p_k^T A \rightarrow p_k^T A (x^* - x_0) = \sigma_k \, p_k^T A p_k$$

$$\Rightarrow \sigma_k = \frac{p_k^T A (x^* - x_0)}{p_k^T A p_k}$$

So, that means how many p’s do I have? I have n, right? So, I have p_0 up to p_{n-1}. This is my set of conjugate directions. And moreover, as we showed, they are linearly independent. So, if I know that they are linearly independent, I want to know how far my starting point is from the end point.

So, if I write a vector like this, x^* is where I end, x_0 is where I began. In this case, is this a vector? Can it be written in the basis of p’s? Very obviously yes. So, I can write this as

$$x^* - x_0 = \sum_{i=0}^{n-1} \sigma_i \, p_i$$

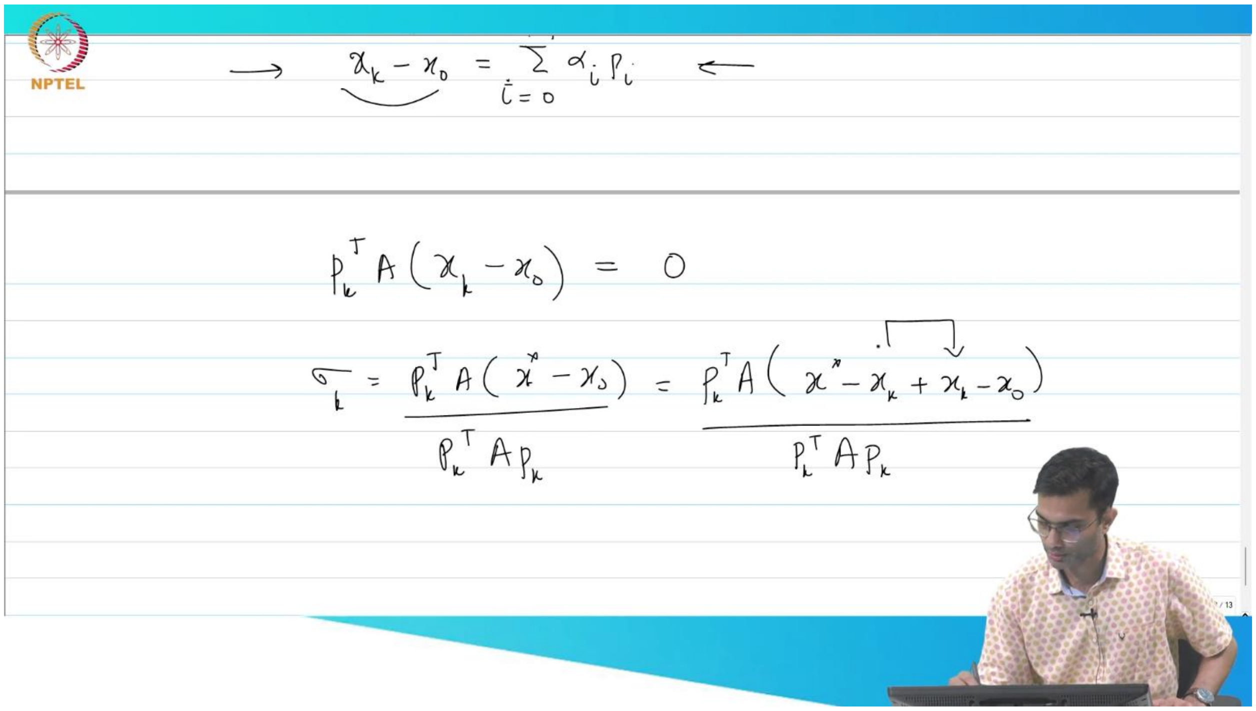
This is just basically a linear algebra basis expansion.

Again, we have one tool, one trick in our box. If I want to evaluate the σ_i’s, what can I do? What is the tool that I have used and we continue to use again and again? Use the conjugacy property, right? If I stick a $p_k^T A$, right, it will annihilate all terms except what? Only one term will survive, right. There are p_0 up to p_{n-1}. If I do this trick of $p_k^T A$, only one guy is going to survive, all other terms are going to go to 0.

So, if I left-multiply by $p_k^T A$ on both sides of this expression, what am I going to get?

$$p_k^T A (x^* - x_0)$$

What is going to be left on the right-hand side? Who survives this? σ_k, right. σ_k is the only guy that survives this. By the way, is this useful? Is this a useful way of calculating σ_k? Common sense answer. Is this a useful way of calculating σ_k? Look at the expression, is it useful? No, because I do not know x^*. If I knew the solution, why should I do all this? This is just a way of manipulating the expression. Obviously, I do not know x^*.



$$\longrightarrow \underbrace{x_k - x_0} = \sum_{i=0} \alpha_i p_i \longleftarrow$$

$$p_k^T A (x_k - x_0) = 0$$

$$\sigma_k = \frac{p_k^T A (x^* - x_0)}{p_k^T A p_k} = \frac{p_k^T A (x^* - x_k + x_k - x_0)}{p_k^T A p_k}$$

So, I cannot evaluate σ_k this way. One mistake over here which is what? There is something missing in this expression. I forgot to write $p_k^T A p_k$. So, the expression I got for σ_k was:

$$\sigma_k = \frac{p_k^T A(x^* - x_0)}{p_k^T A p_k}.$$

Now, I notice there is an $x^* - x_0$ over here, right? This is telling me how far I am from the starting point to the endpoint. But I want to somehow bring in the x_k, the position at the k-th step. So, let us try to bring that guy into this expression. x_k was, if I started with x_0, how do I get x_1? x_1 was:

$$x_1 = x_0 + \alpha_0 p_0,$$

and to that, I would add $\alpha_1 p_1$ to get x_2, right? So, continuing this, the last term should be $\alpha_{k-1} p_{k-1}$. So, notice one very simple thing from here. If I take $x_k - x_0$, what am I getting?

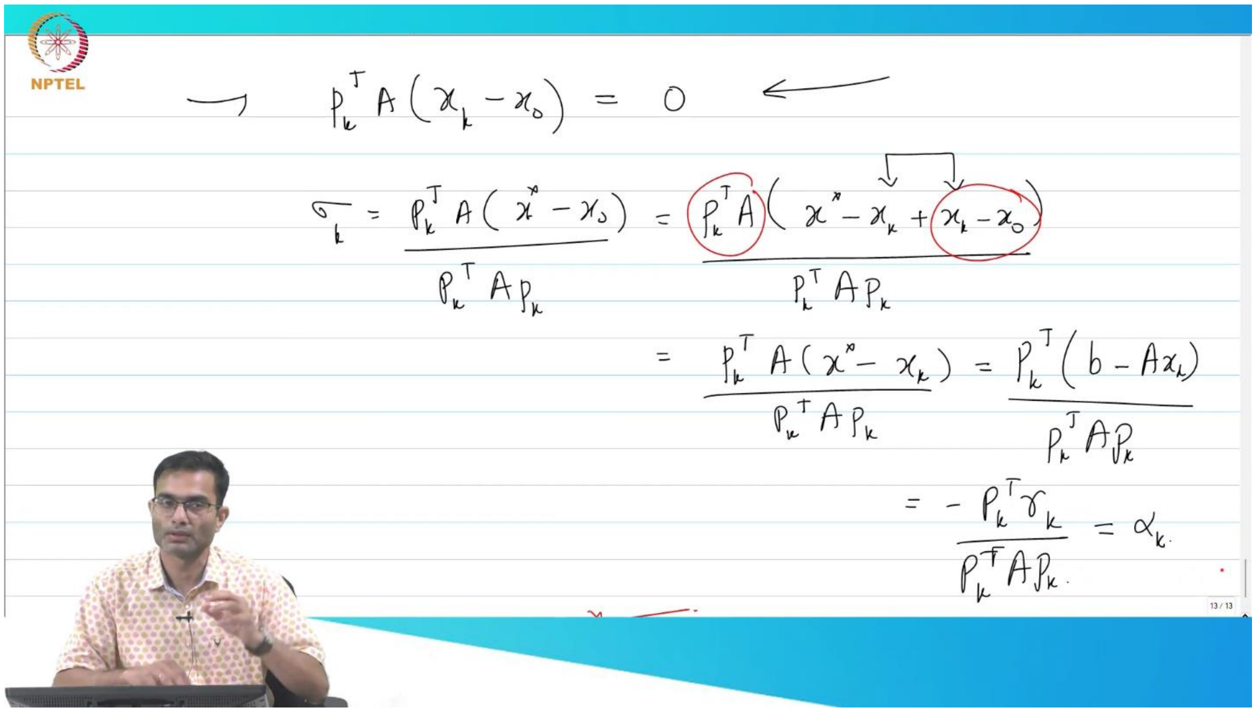
$$x_k - x_0 = \sum_{i=0}^{k-1} \alpha_i \, p_i.$$

So, this distance, $x_k - x_0$, is now written as a linear combination of the p_i's, but not all the p_i's. How many p_i's are coming in here? Only k terms, from p_0 to p_{k-1}. That is giving us a hint.

Now, having gotten this expression, let us apply our usual trick. What is our usual trick? To kind of decimate this, see I do not know these α_i's, but I do know that they were given in the recipe of the CDM. The α_i's were given by exact line search, right? So, I can also estimate the α_i's by using the conjugacy property. If I stick a $p_j^T A$ over here, it is going to isolate one of the alphas. I will get something to compare with.

So, let us quickly do that. I am going to take:

$$p_k^T A(x_k - x_0).$$



$$p_k^T A(x_k - x_0) = 0$$

$$\sigma_k = \frac{p_k^T A(x^* - x_0)}{p_k^T A p_k} = \frac{p_k^T A(x^* - x_k + x_k - x_0)}{p_k^T A p_k}$$

$$= \frac{p_k^T A(x^* - x_k)}{p_k^T A p_k} = \frac{p_k^T (b - A x_k)}{p_k^T A p_k}$$

$$= \frac{-p_k^T r_k}{p_k^T A p_k} = \alpha_k.$$

13 / 13

What will I get? Will I get α_k? There is no p_k left on the right-hand side, so this is going to be equal to 0.

Now, here is the final trick. So, recall this expression for σ_k:

$$\sigma_k = \frac{p_k^T A(x^* - x_0)}{p_k^T A p_k}.$$

What was the denominator? $p_k^T A p_k$. I am simply going to add and subtract x_k. Fine, I just added and subtracted x_k. Can I use the previous result anywhere? This term acts on the previous expression $p_k^T A(x_k - x_0)$, which gives 0. So, this becomes:

$$\sigma_k = \frac{p_k^T A(x^* - x_k)}{p_k^T A p_k}.$$

Now, we know that $Ax^* = b$, so we can write:

$$\sigma_k = \frac{p_k^T (b - A x_k)}{p_k^T A p_k}.$$

This is just the residual, $r_k = Ax_k - b$, so the expression becomes:

$$\sigma_k = \frac{-p_k^T r_k}{p_k^T A p_k}.$$

Does this expression look familiar? We have seen it before. What was it? α_k.

So, do you see what has happened now? I have my final distance between the starting point and the end point, written as a linear combination of all of these vectors from p_0 to p_{n-1}, and the coefficients σ_i are nothing but the step lengths I took at each iteration. I did the first iteration as $x_0 + \alpha_0 p_0$, at each iteration, I am going along that conjugate direction by a certain length α_k, and at the end of it, in n steps, I will reach the solution.

Subsequently, we will do some visualizations to help understand this better. To review the proof: we started with a set of linearly independent directions, which means that the error (start - end) can be written as a linear combination. The coefficients are the alphas, and this gives the exact distance covered during each step.

Each time I walk along a direction p_k for a length α_k, in the end, there is nothing left.

Where does "at most n" come from? Can I reach the solution sooner? If the residual r_k or the gradient ∇f_k is 0, then I have already reached the stationary point. So, the "at most n" comes from the fact that once $r_k = 0$, the process terminates. Thus, the method guarantees to converge in at most n steps.

Notice that the only thing I need for this is the expression for the p's. Give me the p's, and this is enough. Do I need to explicitly store the matrix A? No. I never need A in isolation. This expression only requires the product of a matrix with a vector.

In numerical routines, that is a much more relaxed requirement, because storing A requires memory proportional to n^2, whereas the matrix-vector product only requires n entries, which is much more memory-efficient.

So, iterative methods are specifically designed for situations where you run out of memory and cannot do the whole thing directly. It will take more time than a direct method, but if you have no other options, this is the way to go.

So, those of you using commercial software for solving systems, if you see in the log that it says, "using this indirect solver" or "that indirect solver", this is one of the methods in between.

By Gaussian method (Gaussian elimination), we know the complexity is $O(n^3)$, and it is also a direct method. So, if you have enough memory, you may prefer a direct method, as it is faster. If you do not have enough memory, you cannot use Gaussian elimination.

Direct methods like Gaussian elimination and LU decomposition have the same complexity, but they need the matrix to be stored in memory. In an iterative method, you may decide beforehand that you do not want 10^{-16} error. Instead, you can tolerate 10^{-4} error, and thus abort the calculation earlier. This flexibility is not available in direct methods, where you must wait for the entire computation to finish.

This is why, in many big data science applications, iterative methods are preferred because, after several iterations, you can stop when the error is sufficiently small.