Optimization Algorithms: Theory and Software Implementation

Prof. Thirumulanathan D

Department of Mathematic

Institute of IIT Kanpur

Lecture: 47

This is lecture 2 of week 10. Recall from the previous lectures, covering all of week 9 and the first lecture of week 10, that we discussed the simplex method in detail. This is an age-old method for computing the solution to a linear programming problem. You may recall the name Dantzig, who formulated the simplex algorithm in the 1940s. It has been over 80 years since he formulated the algorithm.

We mentioned two issues with the simplex method. One is that finding an initial basic feasible solution could be a hard task. People did improve this part to some extent after Dantzig, on how to find a basic feasible solution initially. We are not going to discuss that since we have other algorithms to discuss. The second issue is not solvable.

It is a problem with the simplex method that in the worst case, it could end up traversing all the vertices. Imagine, for example, if n is 50 or n is 100.

With the exact problem given at the end of the previous lecture, it would take 2^{100} iterations. 2^{100} is 1 followed by 30 zeros, which is huge. That is an inherent part of the simplex algorithm; nothing can be done.

Perhaps the right question to ask is: if the simplex method has this issue in the worst case, that it could take so many iterations and may end up traveling all possible vertices, why did people use it at all? Because in the worst case, it can be as bad as brute force. The reason is that, other than a few pathological examples like the one given in the last class, there are only a few pathological examples like that.

In general, if you take up a linear programming problem, the simplex method gives the answer in very few steps. Of course, there are certain pathological examples, but in most cases, the number of steps the simplex algorithm takes is very few. It is just these pathological examples that have derailed our discussion. But in general, that is not the case. For most examples, the simplex method gives the answer in very few steps.

There were a few other methods, like the ellipsoid method. You can look at these methods if you want. So, other methods or algorithms to find a solution to LP: there are a few other algorithms, and this is one method you would often find when searching for algorithms to solve LP.

The ellipsoid method does not have this issue; in the worst case, it does not take so many vertices. It solves in polynomial steps in n, say n^2 or n^3 or something of that sort. But that is the worst-case guarantee. In general, if you give some linear programming problem to the simplex

algorithm and some linear programming algorithm to the ellipsoid method, you will see that the simplex method, in most cases, or in almost all cases except these pathological cases, outperforms the ellipsoid method.

Though the ellipsoid method has a worst-case guarantee, people were still using the simplex method because in practice, the simplex method gave the answer in very few steps. That is one of the reasons why I am not teaching the ellipsoid method; it is completely outdated. You could say the simplex method is also outdated, but at least for academic purposes, people teach the simplex method.

The reason is that it gives a very nice perspective of linear programming problems and provides many properties of linear programming problems. The simplex method is still used in the literature, but in practice, people have moved to more sophisticated methods. Some of those are like the affine scaling method and Karmarkar's algorithm.

I mentioned that in the 1940s, Dantzig came up with the simplex method. The next most useful method that was said to replace the simplex method was Karmarkar's algorithm. It is popularly called the interior point method; actually, both affine scaling and Karmarkar's algorithm are called interior point methods.

The reason is, if you recall how we find the solution in the simplex method: we start with a vertex and keep moving around different vertices until we converge to the solution. You could call the simplex method a boundary point method because it only traverses the boundary; it never goes inside the feasible set. For example, if you have a feasible set in two dimensions, the simplex method starts at a point, then traverses along the boundary, from vertex to vertex. It never goes to the interior.

You will never see the iterates of the simplex method going through the interior. Suppose the solution is here and the starting point is there; the simplex method always traverses along the boundary, never through the interior. So, the simplex method can be called a boundary point method. Whereas the affine scaling method and Karmarkar's algorithm travel through the interior. You can start from an interior point.

Recall that the simplex method starts with a vertex and moves across vertices. In the affine scaling method or Karmarkar's algorithm, you can start at an interior point. You need not start at a vertex; you can start at a vertex also, but you can also start at an interior point. How do the iterates traverse in the feasible set? They actually traverse in the interior of the feasible set; they do not traverse the boundary points as the simplex method did.

So, what we are basically saying is that this is actually a very different method. Of course, both Karmarkar's algorithm, the affine scaling method, and the simplex method are all solving the same LP problem, but the way they approach a linear programming problem is very different. The simplex method approaches by traversing the vertices; Karmarkar's algorithm and the affine scaling method approach by traveling through the interior. You will see no connection between the algorithm we saw for the simplex method and what we are going to see for the

interior point methods. Keep that in mind; this might be new to you. If you recall how we built up the algorithms for unconstrained optimization, we started with gradient descent, improved it to conjugate gradient, then with a slight variation moved to Newton's method, and improved it to quasi-Newton method.

All the algorithms were similar. When we did constrained optimization, we started with the quadratic penalty method and improvised it to the augmented Lagrangian method. But that is not the case here. For linear programming problems, we started with the simplex method. We are now going to discuss some interior point methods, which have absolutely no relation to the simplex method.

For some trivia: in the 1940s, Dantzig gave the simplex method. Then, Karmarkar, who is actually a graduate of IIT Bombay (he did his B.Tech there), designed Karmarkar's algorithm, which performs better than the simplex algorithm and also gives a polynomial guarantee. It does not take 2^n steps; it takes very few steps and removes the pathologies or issues found in the simplex method. But people had to wait for about 40 years to actually have this improvement. In fact, people were trying to work around the simplex method in some sense. Karmarkar's algorithm is something out of the blue; it approaches the problem in a very different way.

With this amount of trivia, we will move to these methods. We will work with both the affine scaling method and Karmarkar's algorithm. We will start with the affine scaling method. The good part, though I said the simplex method and affine scaling method are very different, is that the first step is common. You have to start by converting the given LP to its standard form. There is no escape from that. It has to be converted to this form: minimize c^Tx subject to $Ax = b$, $x \geq 0$, and the rank of A is equal to the number of rows m. That is the first step; no respite from that.

We will first learn the idea and then go ahead to frame the algorithm. The idea is similar to what we did in gradient descent. In gradient descent, when given an initial point, you find a descent direction, which is just the negative of the gradient of the objective function. The gradient of the objective function here is just c, since the objective function is c^Tx. So the negative of the gradient is -c. So, are we just saying that the descent direction at each step is -c? Wait a minute.

If it were that easy, it would not have taken so long. If you just blindly choose -c, you can end up going out of the feasible set. For example, if you start at this point and -c is in this direction, you might end up going out of the feasible set. So, it does not suffice to choose the steepest descent direction, which is the negative gradient. You have to project -c onto the null space of A.

You should ask: what is so great about the null space of A? I suppose you are familiar with the null space.

The null space of A is the set of vectors x for which $Ax = 0$.

I will denote that as null(A). So, that is the set of all x for which Ax = 0.

As simple as that. So, what I am claiming is that you have to project the vector -c onto the null space of A.

Why the null space of A?

Note that we want each point x_k (x_0, x_1, x_2, ..., x_k) to satisfy Ax = b.

We want $Ax_k = b$ and $Ax_{k+1} = b$ for any k.

If $x_{k+1} = x_k + \alpha_k d_k$, then $x_{k+1} - x_k = \alpha_k d_k$. Then $A\, d_k = A(x_{k+1} - x_k)/\alpha = (Ax_{k+1} - Ax_k)/\alpha$.

But both Ax_{k+1} and Ax_k are b, so that is 0.

So, I can retain the point within the feasible set if my direction d_k is such that $A\, d_k = 0$.

So, if you have found an x_0 such that $Ax_0 = b$ (a feasible point), and you want to move to x_1 while ensuring $Ax_1 = b$, how do you do that?

Choose d such that A d = 0, which means you have to choose a d that is in the null space of A.

Great. So, I will choose something in the null space of A, but it also has to be a descent direction. So, what I am going to do is project the known descent direction. I know that -c is certainly a descent direction, but I do not know if it belongs to the null space; it usually does not. So, I will project -c onto the null space of A.

The idea is to project -c onto null(A). Fine, but how do we project? That is a case where you have learned projection in linear algebra; it should be straightforward. But since even those who have learned linear algebra are sometimes not very well-versed with projections, let me spend about 5 minutes discussing what is the projection of -c onto the null space of A.

In fact, I will derive that. Before that, you should know that the null space and the row space of a matrix are orthogonal to each other. That is, if you take a vector from the null space of a matrix and a vector from the row space of the matrix, they are orthogonal. Any vector can be split into its orthogonal components.

If all this is Greek and Latin to you, understand it this way: consider the x-axis and y-axis, or x_1-axis and x_2-axis.

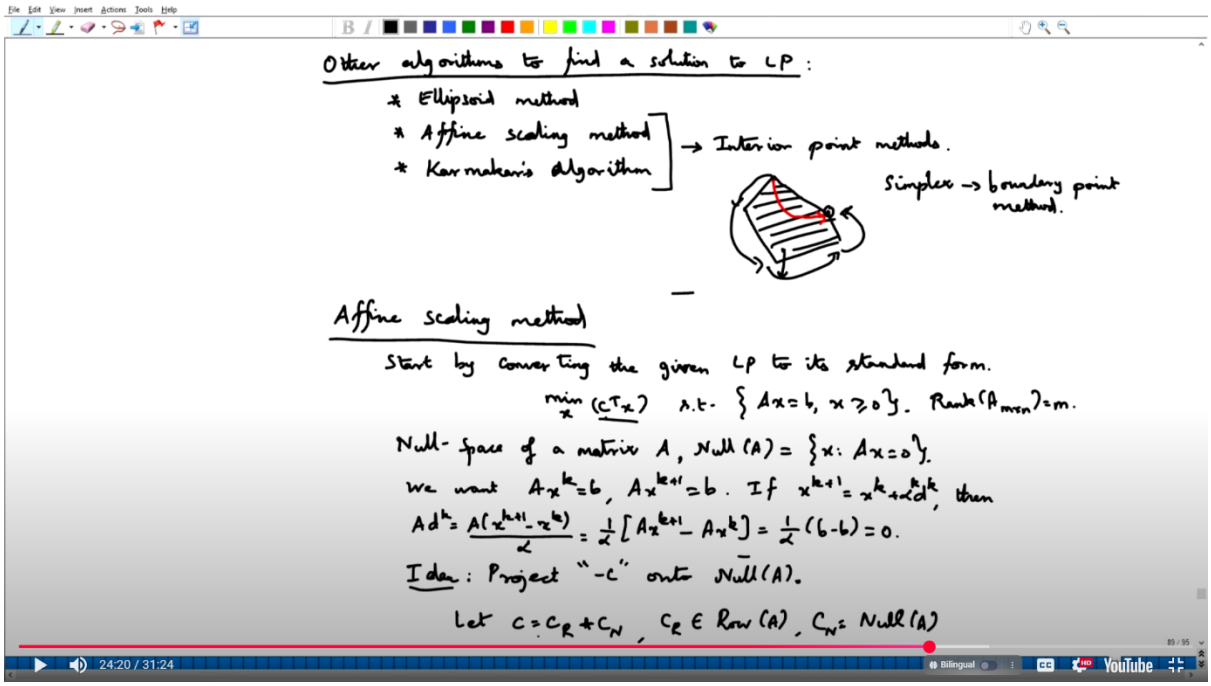
If I give you a point (4,5), you can split it into a component in the x_1-axis (4,0) and a component in the x_2-axis (0,5).

It is not just for the x_1-axis and x_2-axis; for any two orthogonal axes, say the 45-degree axis and the 135-degree axis, you can split (4,5) or any given point into a point in the 45-degree axis and a point in the 135-degree axis. You can do that in n dimensions.

Since the row space and the null space of any matrix are orthogonal, I can split a given vector into the sum of two vectors, where one vector is in the null space and one vector is in the row space of the matrix. So, let me do that for c (we will change to -c at the end).

Let me call them c_R and c_N, where c_R belongs to the row space of A and c_N belongs to the null space of A. The row space of A is all vectors that can be written as linear combinations of the rows of A.

(Refer Slide Time 24:20)



Suppose you write c = c_R + c_N.

Since c_N belongs to the null space of A, it implies that A c_N = 0. And c_R belonging to the row space of A means that $A^T\lambda$ = c_R for some λ.

That is, c_R can be written as $A^T\lambda$. So, if λ is a linear combination, say λ_1, λ_2, ..., λ_m, then λ_1 scales the first row, plus λ_2 times the second row, plus ... plus λ_m times the m-th row. That is c_R. So, you can write c_R as $A^T\lambda$.

Now, the question is: can I find λ? Of course, I can. To find λ, I will write A c_R = A $A^T\lambda$. I claim that A A^T is invertible.

Why? Because the rank of matrix A is equal to m. A A^T will be an m × m matrix, and since it is the product of two rank-m matrices, you will get a rank-m matrix as well.

An m × m matrix having rank m is nonsingular. So, you can invert A A^T. Premultiplying both sides by $(A A^T)^{-1}$, you get $(A A^T)^{-1}$ A c_R = λ.

Now, what we actually need is c_N, not λ, but we will use this to find c_N.

Note that c = c_R + c_N, so c_N = c - c_R.

But what is c_R? If I substitute the quantity $A^T (A A^T)^{-1} A$ here... Let me actually find what is $A^T (A A^T)^{-1} A c$. Instead of c, recall that c = c_N + c_R.

So, you have $A^T (A A^T)^{-1} A (c_N + c_R) = A^T (A A^T)^{-1} A c_N + A^T (A A^T)^{-1} A c_R$.

But A c_N = 0, so the first term is 0. The remaining is $A^T (A A^T)^{-1} A c_R$. Note that $(A A^T)^{-1} A c_R = \lambda$, so this becomes $A^T\lambda$, which is c_R.

So, $A^T (A A^T)^{-1} A c$ actually gives the projection of c onto the row space of A.

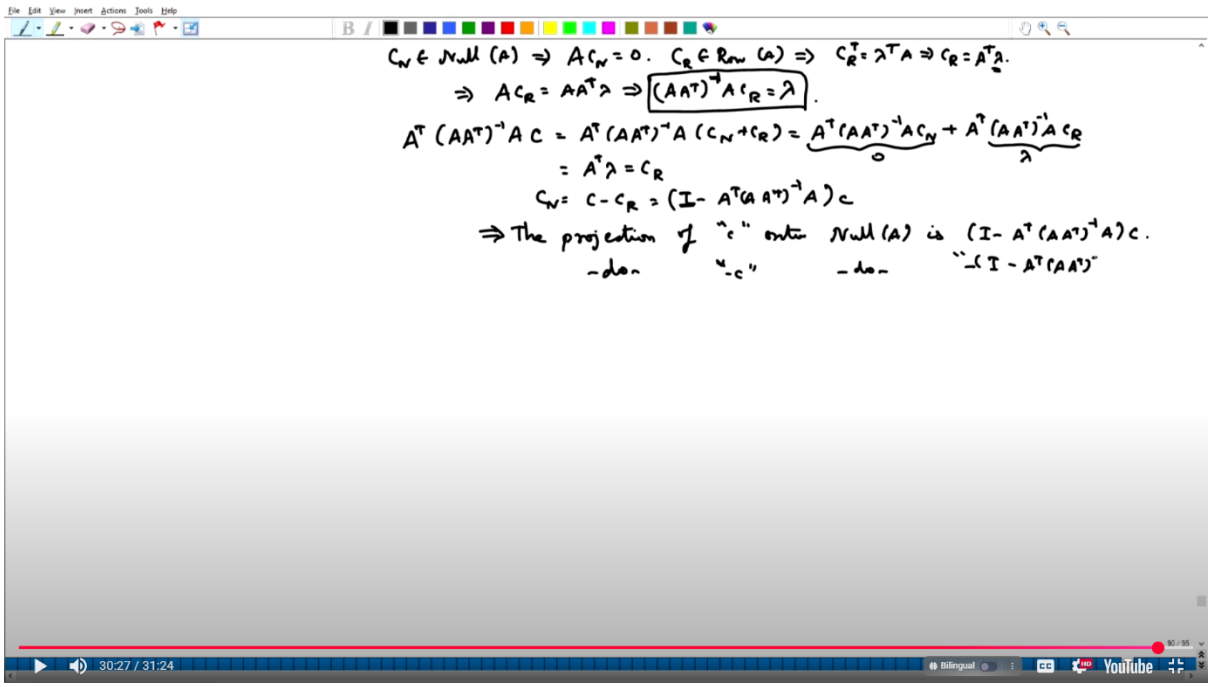
To get the null space component, c_N is just c - c_R. So, $c_N = c - A^T (A A^T)^{-1} A c = (I - A^T (A A^T)^{-1} A) c$.

So, when you want to project a vector onto the null space of a matrix, you just premultiply the vector by the matrix $(I - A^T (A A^T)^{-1} A)$.

So, the projection of c onto null(A) is $(I - A^T (A A^T)^{-1} A) c$.

What is the projection of -c? I just replace c by -c: $- (I - A^T (A A^T)^{-1} A) c$.

(Refer Slide Time 31:30)



So, what we have done until now is: when you are given some x_0, you want to choose a particular direction such that it is a descent direction and you will stay within the feasible set. What you should do is very simple: choose the direction as the projection of -c onto the null space of A.

That is, you need to choose - $(I - A^T (A A^T)^{-1} A)$ c. We will continue the algorithm in the next lecture. Thank you.