# ConvexOptimizationII-Lecture10

**Instructor (Stephen Boyd)**:Today we're gonna do – last time we finished up a bunch of abstract stuff about decomposition, but today we're just going to look at two or a handful of applications in details to see how decomposition actually works, or where it's actually applied. So the first one we're gonna do is rate control in a network. Actually, this is sort of a big topic right now, so if you were to look at this stuff, you would, if you were to go to Google or something, you'd find zillions of papers on this topic. Is that bounce in your – what's that?

**Student:**[Inaudible]

**Instructor (Stephen Boyd)**:Oh, it's good for you. The monitor was just confusing me. That's okay. So let's just jump in and look at rate control. So rate control. This is a very, very basic problem. It's non-dynamic; it's completely static here. So the question is, how do you assign rates to a bunch of flows in the network? So here's the way it's going to work. We're going to have n flows. The route's fixed and they're not going to be split. Although, as you'll see in a minute, it's going to be fine to do multicast or anything like that. Even split flows, that sort of things. But this is just to make it simple. Once you understand how to do this, you can understand how to do all these extension. So the variables are going to be the flow rates on a different flow. And then each flow rate is going be associated with a concave increasing utility function. So that tells you how much benefit you derive from flowing at a given rate. These could be all sorts of things, the utilities. I'll talk about them in a minute. The traffic on a link is the sum of the flows that pass through it. So these are flows. They flow on links and multiple flows go through the same link. You add up all the flows on that link and that gives you a total of traffic. And the typology of the network is given by a routing matrix. So Rij is one if flow j, that's the second index, passes over link i and zero. And the link capacity constraint is the traffic. This is a vector of traffic on the links has to be less than the capacities of the links. So this is the limited resource right here, the edge capacity. By the way, once you understand this, you realize that this exact framework here will instantly be multicast. So in other words, if you have a flow that passes over a bunch of links, splits, and then goes out to a tree, which would be the multicast thing, this handles it instantly. It'll also do split flows, too, if you care about it. You can do things like put a number less than one here.

None of this matters. So here's the rate control problem. You want to maximize the total utility derived by running the flows. That's going to be this. And that's separable subject to respecting the traffic, the capacity limits on your edges. So that's the picture. It's a convex problem, obviously, because this is concave and this is a set of linear inequalities. So it's obviously a convex problem. In fact, the interesting part is here, since we're taking the objective is completely separable. In fact, that's the hint. When you look at a problem, the things that should start lighting up the part of your brain that's associated with detecting structure – by the way, you'll see it's the same part of your brain that does structure as does targets for decentralization. So let me just ask you a couple questions for fun here. If you were to solve this by an interior point method, how would that go? Just roughly how would you write an interior point method to solve that? Let’s start with the objective. What is the Hessian of the objective look like?

**Student:**[Inaudible]

**Instructor (Stephen Boyd)**:Why block?

**Student:**[Inaudible]

**Instructor (Stephen Boyd)**:So diagonal. It’s diagonal. So that should get your attention instantly. And why? Because it’s separable. So immediately, you should look at this and separability should light up two parts of your brain, the part associated with decomposition, but really the same part of the brain. And the other part is structure in the Hessian. So the Hessian’s diagonal. And of course, they’re coupled by the constraints. So what that should do when you see something like that, where the objective is separable, but they’re coupled only by the constraints, is things like dual decomposition, primary decomposition just come to mind immediately. So let’s look at dual decomposition. You’d form a Legrangian. So here you form a Legrangian. And what we’re gonna do is there’s two ways to do maximization problems. One is to keep in mind everything switches or you can put a minus sign in front and make it a minimization problem. So I guess that’s what’s done here. The Legrangian for minimizing minus U is minus U, the objective, plus lambda transpose Rf minus c. And these are going to be positive here. Now of course, when you do this, a very cool thing happens. You write this as R transpose lambda quantity transpose f. That’s this thing here. That’s the jth row of that. And then you have this minus c that comes out over here. So here, these lambda’s are associated with the link. Each link capacity constraint. So lambda 4 is associated with the fourth link. And the capacity constraint. And in fact, we’re gonna see – it’s gonna be clear, anyway, but we’re gonna associate lambdai with a price per unit flow. So it’s a tariff or a rate. It’s a price rate for using that link. And in fact, we can actually interpret all sorts of other stuff as this. Rj transpose lambda, R is a matrix that has a zero and one and tells you what links connect into which. So R looks like this. It’s got here, these are the flows.

And then here you have zero’s and one’s like this to tell you how if this is link one, this row, and then it’s got one’s in the entries, if that flow passes over that link. So this is that. This is R. And if you look at R transpose lambda, it basically says now you just transposed this thing and you go down the column instead of a row. If you look at a row here, you’re looking at a link. And the one’s tell you which flows are on that link. If you look down a column, this tells you exactly a flow, actually. Because, for example, the first column is a bunch of one’s and this gives you the first flow. It tells the list of links that flow one goes over. So that’s what this column is. And if you take this column, transpose lambda, that’s actually a beautiful thing. It’s exactly the sum of the lambda’s over the route taken by that flow. By the way, those lambda’s have an interpretation of a rate. And then you can even interpret this beautifully. This right here, this thing, that’s actually the total rate, because if you go over a link that costs you, whatever it is, $1.00 per flow rate, per megabyte per second, and you flow over another one that’s $2.00, another one that’s $0.50 or whatever, then altogether that’s $3.50 per megabyte per second or whatever. That’s what this thing is. So this is actually the total tariff per flow rate on that route. So this is if you’re charged by these carriers to carry your traffic. That’s what this is. This is, of course, the negative utility. Or if you want, you could flip the sign on it and say it’s the utility minus this, and then it makes perfect sense. This is the amount of utility you derive from having a flow rate fj, and this is the net utility, because you subtract off the cost of running your flow over all those edges. So this idea of the lambda’s being price, it’s obvious for anything – that’s exactly what is it. And this is negative net utility. I want to back up and before I go into this, I did want to mention a few things like what the rate functions might look like – not the rate functions, the utilities. So let me say a little bit about that. They have to be concave and increasing. So here’s your flow. And then here’s your utility. Simplest utilities are something like this. That basically tells you something like it violates we’re assuming here U is strictly concave, but let’s just ignore that for the moment. This utility just says the more flow you get, the happier you are. And getting another ten kilobits per second, even when you’ve already got a lot makes you just as happy as the first ten kilobits per second.

So here there’s no satiation effect. So you don’t – the marginal utility of additional flow is constant here. So that’s this one. This, by the way, might be a good model if that flow represents email or something where you’re just paid by the amount of stuff that kinda goes through. The more the better. But you can have all sorts of other things. You can have this. That’s a very interesting utility. What does this utility say? This basically goes up to some f0. You’re happy if you get to f0, and then above that it doesn’t matter to you. And of course, you can do lots of other things. You can do things like this. That says with a utility like this, you’re basically saying, look, the more flow I have, the happier. I really appreciate flow at whatever this slope is, up to some amount of flow here. At this point, I sort of get sated, and this says I’m still happy with more. But the marginal utility goes down. So this is one picture. You can also do this. You can have something that goes to minus infinity like that. And that says I will get infinitely unhappy if you fail to provide me with a minimum flow. Actually, this is kinda stupid, because if you have a minimum flow on a flow here, remove it, and just decrement the capacities of all links it goes over, and the problem remains the same. But still, this is the idea. So these are pictures of flows. Here are some that people use that are very common. One is a log. That’s extremely common, a log utility. And this would come up in networking, all sorts of other area. And what’s interesting about this one is it says basically what’s equally attractive to you is a percentage increase in flow. So every time the flow you get goes up by 10 percent, it makes you equally happy. That’s going from one kilobit a second to 1,100 bits per second, one megabit per second to 1.1 megabits per second, equally happy. That’s log.

And there’s all sorts of other things in between, square roots and all sorts of things. That’s the picture on the utility. Okay. I just wanted to mention that. So here’s our Legrangian. And the thing to notice about the Legrangian now is this thing is separable. And of course, when this is linear here, and this will work for linear constraints, of course, whenever you have linear constraints and you make a Legrangian, everything splits. So you have perfect – this is completely separable. If I ask you to minimize this over the vector f, it can be done at each flow completely independently. It can optimize itself, because it's a sum of functions of the individual flow rates, and therefore, it can be done completely separable. And it's actually quite interesting. If you think about what actually happens when each flow minimizes this function – or maximizes Uj minus that, that's the net utility, is you're basically doing this. You're publishing prices on all the edges, on all the links. So this link, or tariff, you're publishing prices on the links. And then you tell each flow, you know what, don't worry about the link capacities, but charge yourself, actually really or fictitiously, a certain amount for your flow rate on every link. And then just optimize your net flow. And that's how this works. We'll get to these. So the dual is easy to work out. You just minimize this over each fj. That's a single variable problem here. And of course, you can write this. You recognize it. It's a minimum of a convex function plus another thing. And that can be written in terms of a conjugate function easily. So it's negative Uj, that's a conjugate function. That's, of course, the conjugate. That's convex. That's the conjugate of it. And it's evaluated at minus Rj transpose lambda. So you get this. And the point is, this things here, these things are just scalar functions.

So you get the following dual rate control problem. And actually, the interesting part is the primal rate control is from the point of view of the users. Basically says calculate flows that won't violate the resources available on the network. That's the edge capacities. And make sort of the sum of the happiness of everyone operating on the network maximized. The dual problem is this one, and the variables actually are the link prices. So the dual problem – this is often the way it works with primals and duals, as you know now. The dual problem actually takes the other point of view, so you could argue the dual problem is actually the one faced by whoever's running operating the network. Basically, the dual problem involves prices, and it says how should the network price all the links? You're gonna price them and you're gonna price them so that actually the users are gonna back off and you'll come in under – if you adjust your prices right, they'll come in under the capacity. So these are to be interpreted at link capacities. This is a pricing problem. And what you want to do is you want to price the whole thing in such a way that you minimize the net happiness. Sounds kinda evil. That's actually exactly what this is. How do you get a subgradient of the negative of this, which is, I guess, lambda transpose c plus? How do you get a negative of this? It's very simple. It's stuff we've looked at now several times. It's this. If you have a flow, you have some lambda's, you evaluate the optimal flow under this pricing. Optimal means that flow optimizes its net utility. That's fj here. And you optimize this and you go back here and you look simply at R (f bar), that's the actual traffic on the network, minus c, and that's an element of the subgradient set for this dual function, the negative dual. So that's what that is. And this makes perfect sense. This thing here is actually the traffic violation or something.

If it's positive, it basically says that the pricing – well, if it's positive, it tells you that the subgradient of this thing, if you increase the price, something's going to – whatever it is. It's all mixed up now, because I'm maximizing and minimizing and squished it all around the middle. But that's the picture. I should add, by the way, g is strictly convex and differentiable. Sorry, if the U's are, this is actually differentiable. But I'll still write this, because then I'm gonna cover cases like where you have kinks in the utility. By the way, if you have a kinked utility that goes up linearly, and then transitions to a new slope, what do you think the optimal solution is going to look like? What would you expect when you solve a utility? On a network, a utility looks like this. Like that. Let's suppose that's one. What do you think?

**Student:**It'll be bunched up or the derivative is discontinuous.

**Instructor (Stephen Boyd)**:Where else are they gonna be bunched up?

**Student:**Zero.

**Instructor (Stephen Boyd)**:Yeah. Look. This is like your old friend the exact penalty type thing here. And then that's another kink. And so, in fact, if you optimize a network with, whatever, 10,000 flows and 50,000 edges, when all the smoke clears, we expect the following to be true. A bunch of flows will be sitting right at one. Some might be out here, because there might be some light traffic areas where it's not really congested. If a flow goes through a really congested area, or multiple really congested areas, where do you think that optimal flow's gonna be? What do you think? Zero. That's this thing. And by the way, if that happens, then you can go back and brag and say things like, "Wow, I'm using convex optimization to do admission control." Because that's actually admission – if the optimal flow rate is zero, it says, actually, any amount of flow I allocate to you will lead to a decrease in the public good. That's the sum of the utilities. You're too expensive. And that would be a flow that goes through – if you go through a whole bunch of congested areas, then you're not worth admitted to the network. So that's admission control.

**Student:**Does it have to do with the fact that it's a linear program?

**Instructor (Stephen Boyd)**:No. It has to do with the fact that the slope here is – so if I did this, like that, and then these are both curved, it has to do with the non-differentiability here and here. So this could be x to the 0.8 over here, and then this could transition to x to the 0.5. You'll get exactly the same thing. So it's the kink that matters there. So here's dual decomposition rate control. It works like this: you start with an initial link price vector. For example, you say, look, everybody is $1.00 per megabit per second. Initialize that way. It doesn't make any difference. Then it says, each route, you sum the link prices along the route, that's calculating this thing. And then that's the total price on flow j. So each flow walks along – you can even imagine some protocol for this. This is obviously not how real flow control protocols work. But actually, a lot of people have done a lot of work on back interpreting real flow control protocols, as in TCIP, back to this framework. But nevertheless, you can imagine a protocol where you go along the thing and as your packet moves through across each link, somebody, like the operator, goes into the header somewhere, finds a price, and then increments it by its price. Then it gets to the end, and then it returns, you look at the header, and it tells you the total price on that route. So this says, collect the prices on the routes, and then it says you optimize the flows completely separately. So this is completely independent. So every flow just independently optimizes itself. And what is does is it maximizes what it really wants, that’s this utility, minus, and then this is either a fictitious or it could absolutely be a real charge, and it subtracts the charge for using f. That’s what this is. This makes perfect sense. What would happen here if one of these were zero? What’s the fj then?

**Student:**[Inaudible]

**Instructor (Stephen Boyd)**:[Inaudible] plus infinity. No problem. These are increasing. And if it’s costless, you’ll just say, not a problem. And you’ll open up your flow control, you’ll open up your valve to the widest thing it can be.

**Student:**Are there routers that make a decision which way to send the packet?

**Instructor (Stephen Boyd)**:No. In the problem we’re looking at right now, the typology, completely fixed, the routes are not changing. The next thing we’re gonna look at is gonna do exactly that. Could you combine them? Yes. Once you get the idea on these things, you can solve any of these problems. Routes are absolutely fixed. Everybody got this? So that’s the picture. By the way, there’s no reason to believe that when you set the flows this way your link capacities will be respected. There’s some obvious things you can say. Let me ask you this, what if you get this to step three? Step 3 says these are basically flows calculated with no direct knowledge of link capacities. It doesn’t tell you whether you’re over or under or anything like that. Just says that’s the price. Please deal with it. Please crank your flow us until the marginal utility you get is equal to the price. That’s all you’re doing is getting U prime equals lambdaj. Then you calculate this and let’s actually just figure out some things here intuitively. If s turns out to be quite positive, what does it mean? It’s a slack on the link capacities – they’re all positive and they’re all substantial. What does it mean?

**Student:**Lower prices.

**Instructor (Stephen Boyd)**:Yeah. It means that the flows are not large enough. The flows are not large enough because the prices are too high. So it basically, you told them some ridiculous price, they all said, “Oh, that’s expensive,” and they all backed off to some slow rate, just dribbling stuff down the network. And you’re way under utilized. So in fact, if this is positive, it says that the prices should be lowered and good. By the way, it’s a 300 level class, if the signs came out wrong, it wouldn’t bother me in the slightest. That’s your homework to figure out. But the story still goes. So in this case, what would happen is this is the dual subgradient update. It says take the old prices, subtract from that some positive multiple of the current slacks, and then you take the plus part of that. By the way, can a link charge, a lambda, can it become zero? If this becomes negative and you take the plus part, wouldn’t this become zero? Is there a pathology there? When you solve an optimal flow problem, can an optimal lambda be zero?

**Student:**[Inaudible]

**Instructor (Stephen Boyd)**:Sure. No flow. Any other cases?

**Student:**If you're gonna pay people to use it.

**Instructor (Stephen Boyd)**:We're not allowed to do that. Lambda's gonna be positive.

**Student:**[Inaudible]

**Instructor (Stephen Boyd)**:When would that happen?

**Student:**[Inaudible]

**Instructor (Stephen Boyd)**:You means flattens out? This is simple. If at the optimal flow solution, that link capacity has positive slack, then the optimal Legrang multiple is zero. That's all it is. It just tells you that. So how could you get a link that is – in fact, someone tell me, what is the simplest case? Show me a case where a link capacity is not active at the optimum? It doesn't sound right, right, because the more flow you get, the more – can someone give me an example?

**Student:**If one link with high capacity was surrounded by links with a low capacity?

**Instructor (Stephen Boyd)**:There you go. Perfect example is just – simplest one. Your example was exactly right. But here's the simplest one. The simplest one would be this: two links like that. This one has a capacity of ten. And this one has a capacity of one. And this is one flow, like that. Done. So obviously, the optimal flow is one here. I don't care what the utility is, as long as it's increasing. Optimal utility is one. So the link capacity here is never active. And therefore, the Legrang multiplier on this one is going to end up being zero, and it's going to be completely determined by this one. So that's the picture. Okay. So these are fun. It's like all dual decomposition algorithms, they always fit a story. In other words, you can explain this to someone, you can explain this to undergrads, you can probably explain it to high school students easily. And then you might ask, so why did you sit through 364 and various other classes to get here? What would I say. I'd say, "You know more. You have a more sophisticated view." Or, "There'd be no reason to believe that that method would work, and now you know otherwise." You know that it would work. Is that really worth sitting through all those classes? I'll have to think about that. I'll get back to you on that.

**Student:**Is it possible to have a price [inaudible].

**Instructor (Stephen Boyd)**:Yes, it is. So I'm showing you the simplest possible dual decomposition method, straight subgradient. You could do all sorts of others. You could do a two-term recursion and update lambda depending on not just the current thing, but the last one. These things would work immediately much better. And in fact, for this particular problem, a lot of people have looked at you write the lambda's as the x of some other variables, and you update those. And that leads to a multiplicative update. And I'm just trying to think. There is one minor hole in this, but it doesn't matter. What can happen is if you started with a two high step size here, then here's what would happen. If you start with lambda large and alpha k large, lambda's large, so what's gonna happen is you’re gonna be way underflowing. Your surplus is going to be high. But alpha’s large, so it means that in your next step, all lambda’s are going to go to zero, and now everybody goes insane at the next step, because everybody says, sorry, it’s costless on my whole route. Not a problem. And you open up your pipe to the maximum rate. So that’s the only hole in it. I’m not gonna do all the details to fix that. You can do all sorts of things there. You could put upper bounds on the flows or whatever. Doesn’t matter. That’s it. It’s actually a quite beautiful method. And you can see this is very far from the silly little things we were looking at before, where you had the two problems and the two subunits of a firm, and they’re sort of setting the prices and each optimizing separately. This centralizes across huge, vast things. I might add that things like this can actually be done and actually are done. So this is kind of obvious, and I’ve already emphasized it, but the idea is this is completely decentralized, utterly decentralized. The link only needs to know the flow that passes through them.

And the flows only need to know the prices on the links that they go through. So you can easily imagine, for example – not that this would do anybody any good – but you can imagine making an object-oriented implementation of all this, where very little information is available to anyone. In fact, each flow needs to know absolutely nothing except what is the current price. What’s the sum of the prices on the links it flows over? So flow doesn’t even need to know, for example, how many links it has. Totally irrelevant. All it needs to know is what’s the current total price. Each edge does not need to know anything. Doesn’t need to know how many other edges there are in the network, doesn’t need to know typology, doesn’t need to know the id’s on the flow. Doesn’t even need to know the flow’s going through it. It only needs to know how close is it to capacity. That’s how decentralized this is. Okay. Now the iterates, actually, are often infeasible. But in the limit you do have this. There’s lots of ways people sort of deal with this. By the way, there’s something that people do that’s a bit confusing, but it’s widely done, and it works like this. They can fuse iterations with actual time. So here, in the subgradient method, the iterations don’t actually flow at these things. So in fact, probably the best way to conceptualize this is to imagine tiny little packets that have no payload, just like header, and they fly along the route and are tagged. Each edge tags it with a weight. And it comes back to the process that’s going to run the flow. And then each edge is adjusting the price. And this goes back and forth and back and forth until basically somebody, somewhere, I guess there’s an all clear thing, Each edge finally says something like, all right, you’ll never actually get under capacity, but you’ll get close enough.

At that point, a little token goes back to every flow that says, we’re cool and we’re good to go. And then you turn on your flow. Everybody see what I’m saying? That’s the right way to conceptualize this. Unfortunately, a lot of people imagine the iterations here to actually be done. So basically, what happens is, at each step you have the things, when you try a step in a subgradient method, it really means pump that flow down that tube. What happens, then, is actually kind of interesting. And it can be interpreted all via dual decomposition. What happens is this: if you pump too much down a link in this model – not this one, and I think this is the one that leads to all sorts of very strange and confusing things – if you pump too much flow, what would actually happen in a dynamic situation if I have a link with capacity one, but the total flow going through it actually is 1.1? What would happen in a real network. It depends, actually on exactly what the routers or whatever is supposed to do. One thing it could do is just take 10 percent of the packets and just dump them, just terminate them right there. The other thing is it could push onto a buffer. And if it pushes onto a buffer, then it's basically if you're asking for a flow of 1.1 down a pipe but it only has the capacity of one, you pump at one, and your buffer is growing at a rate of 100,000 packets or bits per second. Everybody see what I'm saying? So then it turns out you can actually go back in that situation and you get a two-term dual subgradient thing. And this is this whole class of algorithms now in network flow control called back pressure. And that's exactly what this is. And actually it's nothing but a dual decomposition method. I think I just confused everybody by warning you that there are interpretations of this that are deeply confusing.

Is anyone not confused, because I can confuse them separately. Good. We have stationary point in. It's not like seven pages long, is it? It shouldn't be. Next. What do people think? Let me try one thing, then. That's nice. We can see the weather. That's good. It's sunny outside. That's always good to know. Am I missing – let's just see. Did this get truncated? I don't know what happened here. I thought so. That's – I'm sure you know, there's several things that scare professors. That's one of them. The other one is when someone runs up to you and says, "My God, you're teaching in half an hour." And you're like, "I am? What class?" And it's like, "Music history." And you're like, "No, no, that's a mistake. It's impossible." And they're like, "They're all waiting there." Anyway. This is the other one, if you walk in with the wrong lecture notes and it's three pages long. Here we are. Now let's see if everybody's happy. Look at that. When you do this, as I said, what's happening is only the limit to your flow's respect the capacity constraints. But there's ways to produce a feasible flow if you have to from a proposed flow. And there's tons of ways to do it. One is you could do proportional back-off and you can imagine a whole industry and thousands of papers to be written on this. And so it would work something like this, you could let etai be the traffic on the edge divided by the capacity. That, of course, should be less than or equal to one. But if it's less than one, it means you're under capacity. If it's bigger than one, it means you're over capacity. And you could do the following: if there's a proposed set of flows, that's the current one, and someone says, enough of this silly decomposition nonsense, it's time to actually put some payload in your packets and start flowing, then what you would do is this.

This is a very simple method for generating a feasible flow. It goes like this: you take the flow and you do a proportional back-off, you go over all of your links here. If the worst case – this is like the overload factor on your link. So you find the worst overload factor on your flow. By the way, if that's 0.8, that's great news, because then this is telling you, please crank your flow up by one over 0.8, like that. If this is like three, it means you're way, way off. At least one edge is oversubscribed by a factor of three, and it tells you to back off. So this clearly produces a feasible. That's a feasible flow. Totally obvious. And in fact, this is actually a step in primal decomposition for this, but that's often the case in these things. So let's just look at an example and see how this works. So I have ten flows, 12 links. I should add, obviously, this is not for that. That's totally obvious. I should also add, just to make sure this is completely clear, if you need to solve a flow problem with 1 million flows, 2 million edges, 5 million edges, whatever, by far the best method would be an interior point method. Period. That's by far the best method. By far. So the whole point about this dual decomposition is – that's if you could collect all the data in one place, you'd be way better off doing it that way. The nice part about the dual decomposition is it's got all sorts of data hiding and bits and pieces are encapsulated and all that kind of stuff. That's the advantage, not speed, not quality of solution, nothing else, just the decentralization. Okay. So this is just a simple thing. I guess there's 12 links, or the link capacities are randomly chosen, and we'll use the log utility. By the way, if you go to Google and type in "log utility network," you'll find tons and tons of stuff. And a there's some arguments you could make, like it delivers proportional fairness, or something like that, to everybody. By the way, if you put log utilities, you don't do admission control, and that's kind of obvious. Because the utility associated with denying somebody entry to a network, a flow, setting a flow to zero in a log utility is minus infinity, which is basically means infinitely uncool and so it doesn't happen. Everybody gets a little bit in a log utility type thing. So the optimal flow, you can just get a formula for this. It's very, very simple. You put a log minus this and it's a conjugate of a log.

I forget what is it, but it's something explicit. And set all the initial prices to one. And it will take a constant step size. Actually, this, of course, would not work or would work in a limited sense if these things were non-differentiable. In turns out, in this case, the dual function is completely differentiable, in which case, what you can say is this: if it's constant and small enough, you're guaranteed to converge. That's the statement for differentiable case. And in this case, it's differentiable. Okay. Here's an example. These are iterations running. And this shows you the utility derived here by the current thing. And this shows you g of lambda, which is an upper bound on this utility. So that's what's happening. And you can see here that in something like 50, 60 steps, you've gotten the utilities about the right way. Oh, this is the utilities of the corrected ones, where you back off here. So that gives you a primal point, primal feasible, and that gives you a dual feasible point, which is the upside. Now as you're running, you run a maximum capacity violation. And that looks something like this. I guess when you first start this violation's as big as by one, or something like that, roughly, 0.7 or 0.8. Something like that. Now, all the capacities themselves are between 0.1 and 1, so we can get a scale for this. It says that already, by about 20 steps, your capacity violations are pretty small. And certainly, by the end of this, your capacity violations are ten times smaller than the smallest capacity edges on the network. That gives you the rough idea. So now we're gonna do – that's odd. That's very odd. Now we're gonna do a different problem. And these are just supposed to be two different problems in networks and stuff like that. And actually, you'll see that once you get the ideas, you can do zillions of generalizations of these and you can combine them and all that stuff. But these are sort of the two prototypes of these things. So this single commodity network flow, single commodity means it's something you should think of a flow or traffic on an arc. A single commodity means that you should think of electricity, water, gas, diesel fuel, something that's interchangeable. And you don't mind what it is. By the way, in a communication network, a single commodity flow, that would be just packets all destined for the same place, or something like that.

That would be a single commodity flow. So it's basically like a sensor collection network.

Then you have single commodity flow, something like that. So everything is the same, it's all interchangeable and so on. Okay. What you're given is an external source of sink flow at each node. And the sum of the external sources or sink flows is going to be zero. We'll see in a minute that that has to be. And we'll have a node incidence matrix and this is the graph incidence matrix. It just tells you which edge goes into and out of each node. And it's the traditional one, where you have the rows give you the edges and the column index, j, gives you the – did I get that right? No, nodes. It's nodes by edges, like that. Flow conservation says this: it's Ax plus s is zero. So if you look at Ax the vector, let me just – I shouldn't have to – let's just write it down to get the picture. So we've got a bunch of nodes, like this. And you have directed edges. And then each one has an external flow into it. By the way, these edges do not mean that the flow has to go from one – it's simply a reference direction, like in a electrical circuit. So when the edge goes like that, that says that if the flow is actually this way, it's positive. If it goes that way, we're gonna call it negative. So it's just like an electrical circuit. You take a branch, you label the arrow, does not mean the current goes that way. That's the reference direction. And it merely means that positive current means you're going that direction, negative current means you're going the other way. Likewise, these sources that enter the network like this, they can be sources or sink, depending on the sign. And if you form Ax – are we using x for the flow? Yes, we are. So if x is a vector of edge flows, Ax is a vector and the height of it is the size of the number of nodes. And Ax tells you precisely the violation.

Ignoring these flows, it tells you precisely the net flow at that node where you add up everybody coming in and you subtract everybody going out. And what's leftover, if you add in the s, it means that all the flow balances. So you can think of this as the basic conservation equation. These would be the exact same equations in an electrical network. This would be absolutely identical. So here the x is the current on a branch, s is an external current injected, and you'd have Ax plus s equals zero, in which case, this would be something like – this would be Kirkoff current law. And what we're going to do is we're going to have – this, of course, describes lots and lots of feasible – there's lots of x's that satisfy Ax plus s. Well, not lots; it depends on the null space of A, of course. So any flow and then you have – oh, by the way, what is an x in the null space of A called? I'm just asking the English. You can almost guess what it's called. What does it mean to say Ax equals zero in a network like this? Let's say x equals zero, what does it mean? I'll draw a picture and then you can tell me what the null space is. We're going to do a super simple network, it's going to have two nodes.

**Student:**[Inaudible]

**Instructor (Stephen Boyd)**:There. I've got two nodes and two flows. And I'd like to know what on earth, in this case, what does the null space of A mean? Give me an element in it and please tell me what it means. What's an element in the null space the way I've drawn it here? Student:

[Inaudible]

**Instructor (Stephen Boyd)**:Any multiple of one-one. Right. Everyone agree? If x1 is equal to x2, Ax equals zero, because there's the net flow here is zero, the net flow there is zero. So you wouldn't be surprised to find out that the null space of A is actually called – any element in the null space is called a circulation. So in this case, it's literally a circulating flow. But in a general case, you'd call it a circulation, too. By the way, it does not have to be a flow that goes along the path like something like this. That does not have to be. Actually, those are in the null space, but you could have some weird thing where there's all sorts of positive – anything in the null space is called circulation. Just to get the idea for how all this works. So there are lots of things in the null space. Another way to say it is this, find one feasible x that's a flow that satisfies your flow requirement, your external flow, and then you're really asking, to that, add something in the null space. So how to you optimally add circulation or whatever to that? And what you want to do is you want to minimize a flow cost function. And if you wanted a picture for this, it doesn't really matter. This can be a wireless network, and this can be the power required to support a level of flow. So in a wireless network, for example, if you say you're transmitting a megabit per second, you calculate on your rate curve what power it takes to do that. And then this would minimize the total power on the network, for example. But that's the idea. So this is single commodity flow network flow setup. The network flow problem, the single commodity flow, looks like this, you minimize these separable flows subject to Ax plus s is zero, and that's convex, readily solved with standard method. I mean, completely readily solved. We'll see the dual. The dual is going to be quite beautiful.

As you won't be surprised, a flow problem here, the dual is going to involve a problem involving potentials. Again, if you know electrical engineering, you would suspect this. This would hardly be surprising. So we'll see that when we work out the dual, the dual variables are going to be potentials at the nodes. And in fact, it's more than that. You're going to find out that all that really matters is the potential difference across each edge. I'm just saying, you should be prepared for this. It's going to happen, but I don't mind saying it now. Again, if someone just walks up to you and says, "Solve this problem," and there's nothing else here, there's no other reason or excuse for decentralization – by the way, the argument that it's cool doesn't really work. It is cool, but that doesn't matter. You'd be way better off solving this using a standard method from 364a. So sparse matrix techniques would just beat anything, if you could collect the data in one place. We'll see a decentralized solution. It's going to look different from the flow one, but you'll get the idea. So we form a Legrangian. That's a separable function of x plus nu transpose Ax plus s. And of course, the constraints are linear, so when you form a Legrangian, everything separates. So this is just completely generic with linear constraints. And now, this function here is completely separable in the x's. Why? Because this is separable, starts that way. And this is linear. Everything linear is separable. Any linear functional – function is separable in all variables. I said linear functional, and actually, that's correct. Linear functional is a linear function that's real valued. And that's actually very interesting, because this says that if I told you what nu is, these Legrang multipliers – by the way, the Legrang multipliers are associated with nodes.

The variables are associated with flows on edges, and not surprisingly, the Legrang multipliers are associated with nodes, because this vector has the height of the number of nodes. And in fact, it really tells you the node – Ax plus s is very simple. It's the vector of KCL violations. It's how much are you violated conservation of flow at that node. This is associated with nodes. And we'll see that they're potentials. We'll see that very quickly. So this is completely separable in the x's. And let's actually look at what A transpose nu is. So the – let me try to get that right. That's gonna be – so we go A transpose nu transpose times x. That's the component in the Legrangian associated with the constraints, the linear component. And this has various entries, but the entries look like this. It's Aj transpose – and that is nu. And the Aj's are columns that have a plus one and a minus one in them exactly. And it encodes for the jth edge the head and the tail node. So that's what Aj is. In the A matrix, it looks like this. You've got each column has a plus one in it and somewhere a minus one. Everybody else is zero. And that encodes that for this edge, it goes into or out of – I forget what the convention is – one and then out of the other. Okay. This thing is beautiful. This vector is the difference in potentials across that edge. So if I have the jth edge and I form Aj transpose nu, I take this thing and I transpose nu. And it if this is nuk and this is nul, then Aj transpose nu is nuk minus nul. So it's quite beautiful. And in fact, you can already see something very, very cool. Notice that the Legrangian in nu has the property that you can add a constant to all entries of nu and nothing happens. I can add 27 to every entry of nu, and the Legrangian is completely unchanged. Is that true? No. Sorry, it is. Sorry, I confused myself momentarily.

Let's see, it's obvious in one case, but not in the other. In one case, it's easy. Every one of these, each of those vectors looks like it's got a one and a minus one. If I had 27 to every entry of nu, it makes no effect on that, because this is calculating potential differences across an edge. Here is the reason it doesn't matter, because 1 transpose s is zero. So that's why it doesn't make any difference here. The sum of the s's is zero. Okay. So we use the notation delta nuj to denote the potential difference across edge j. Actually, these examples are fun anyway, because they should kind of be in 364a, just because you get to see duality and more applications, where you get beautiful interpretations. That's almost always true in duality. Okay. So here's the network flow dual. It looks like this. The dual function is you minimize this Legrangian. And this is completely separable now, because – for each edge, to determine it's optimal flow in the Legrangian, it turns out it needs to know absolutely nothing. It does not need to know anything other than the potential in its head and the potential in its tail. Nothing else do you need to do. And you take that potential difference, you multiply by your flow, and you add in your cost for flow, and you minimize. And you can write that this way: this is the conjugate of the edge cost. And the dual problem is simply to maximize this g. So that shows you how the primal is a flow problem and the dual with a conservation constraint at each node, that's what a flow problem is, so I guess that's okay. And the dual is one involving assigning nodes to potential. So it's like the network flow problem – I mean the rate termination problem, but different twist. How do you recover the primal from the dual? That's easy.

It’s easy only if these are strictly convex, then you get a unique minimizer, xj* of that. And if these are differentiable, you have a formula for it. It’s very easy. And you get the optimal flows from the optimal potentials. And that looks like this: xj* is – this is xj, that’s the conjugate one. And that’s the optimal potentials. Now by the way, this formula is quite interesting. In electrical circuits, this formula is going to be the nonlinear IV characteristic, the current voltage characteristic. Because basically, you think of y here as a voltage potential across an edge. And as result, this function here, the inverse function of the derivative, is the arg min of this thing. This function here is actually the thing that maps an applied voltage across that edge to a resulting current flow. So for example, for a resister that’s linear, and that actually corresponds to quadratic flow potentials, in which case, these are energies. That’s for the electrical analog, if you know what that is. And a subgradient of the negative dual function, that’s easy. It’s actually just this: all the flows totally uncoordinated. There’s no reason to believe you have flow conservation. They look at the potential at their head and at their tail, they privately minimize something, which is sort of a net power, they privately minimize that, and then they say, that’s my flow. And then, basically, each node, you look and you find out what is the flow conservation violation. That’s this. If that’s zero, you’re done. That tells you, of course, that g is maximized. If it’s not zero – this thing here, the flow conservation residual otherwise gives you a subgradient. Okay. So here it is. You do the following. You start with some initial potential vector, nu. And you determine the link flows from the potential differences.

So you simply look at – you calculate this way. Then you calculate the flow surplus at each node. And we’ll call that capital Si. That’s the surplus. And then it says you simply update the node potentials. It’s actually quite beautiful how it works. I will change my story to get the sign right. Just assuming the sign is right here. Let’s try that. So what happens is this, let’s make an electrical current, so you’re flowing from one node to another. The first thing you do is you calculate the node potential difference. And then you simply calculate a flow that goes across that. Then, if the receiving node turns out to have too much current going in, net current, then it says that potential should go up. That’s it. That potential should go up. If that potential goes up, it says that everybody flowing into that node is going to flow a little less in, because the voltage just went up, so you flow a little less in. And by the way, the outflows from that node are also going to increment a little bit. If you pull the voltage on the node up a little bit, the outflows are going to increase a little bit. But if the inflows go down a little bit and the outflows go up a little bit, it says that the total – if that node was over subscribed before, too much net current, then that’s actually going to be a correction move. And the point is just that these are embarrassingly simple to write down. So these are not algorithms you couldn’t figure out or make up intuitively, but the nice part is now you know exactly what they are and how they work. That’s it. So it’s really embarrassingly simple. Well, as I said, it’s completely decentralized and so on. And you get flow conservation only in the limit here. And once again, the same story holds. If you decide to actually run the flows, if this is not just a discussion among edges and flows and adjusting potentials before you turn on the flows, if the flows are actually running, if you have a residual at a node for that step, it increments or detriments a buffer. In an electrical network, that would be capacitors at the edges. And if there’s too much flow for an iteration, no problem, the charge on that capacitor just increments up. It could also be warehouses, where you're storing things or buffers in a network or something. If too much stuff went in at that step, all it does it pop up the queue length, for example. By the way, if too much stuff lowed out, that's capital Si negative, your buffer actually dropped down a bit, so that's the picture. It's probably not a bad idea to explain fully all this, although I think I've said a lot of this. This is related to an electrical network.

And this is only for people who know about electrical engineering, so if you don't, that's fine. So here, the variable is the current in the branch. The source is an external current injected at that node, and they have to sum to zero, otherwise there's no solution of Ax plus s equals zero. I guess that's kind of obvious. This Ax equals s – wait a minute. Did I just reverse the sign on s? I think I did. Wasn't it Ax plus s before? Do you mind just write Ax plus s equals zero. And then we have to figure out if that's injected or pulled out. It's one or the other. Then we'll make it zero, just for aesthetics. The dual variables correspond to node potentials. And then these are the branch voltages. Of course, in electrical engineering, you know – in anywhere, physics or electrical engineering, if you say the potential, by implication that's a quantity where differences matter, not the absolute values. Unless you have some reference or data value, a ground potential. That's the same thing here. And the branch current voltage characteristic is exactly this thing, which is the inverse of the derivative of that function. And what I can mention here is the following: if the local cost flow functions are quadratic, then it turns out this is linear. That's obvious, right, because then the derivative is a linear function. The inverse of a linear function is linear. So in fact, this goes back to 18 – this goes back to Maxwell, who announced that an electrical network, if you have a bunch of flows in a linear resister network, the currents arrange themselves in such a way as to minimize the total energy in the network. When they are minimizing total energy in the network, then almost by accident, you will get flow conservation. That's the other view of electrical engineering. The more primary view is to say, no, there's flow conservation, or something like that. But it turns out it's a minimization problem. That's it. So let's give you an example. It's going to be minimum queuing delay on a single commodity network. So the flow cost function's going to be something like this: it's going to be xj over c minus xj. And this is something like a queuing delay, roughly. It doesn't matter. But this is roughly a queuing delay. And you can see here that as you – it's got a capacity. And as you run the traffic up to the capacity, what's gonna happen is this denominator's gonna get very small and your queuing delay's gonna get very large. So this is that.

On the other hand, if you lightly load it, you're gonna get a very small queuing delay here. This is the queuing delay. And that's, of course, a convex function. And the conjugate of this, you can just work it out, it's not very hard at all to work out the conjugate of that convex function. And it turns out to be this: it's the square root of (cjy minus 1)2 over some range and so on. Actually, I think we have some plots of these. So here are some plots. Here's the queuing delay. It says if you lightly load the network, then the queuing delay on that edge is going to be small. It's going to increase rapidly as you approach – this is with c equals 1 – as you approach the capacity, your queuing delay's going to go nuts, like this. And the conjugate looks like this. It's zero out here, and then it goes up this way. So that's the conjugate. Okay. So this is the mapping from potential difference to you take the inverse of that function before and you get exactly this. It's the mapping from a potential difference to a flow, something like that. By the way, the potential differences here are actually going to be kind of interesting. If you think of the – we'll be able to see that this analogy is going to work beautifully, because if we measure the potential – if you set everything up right and scale time properly, you get the following: your dual variables are actually going to be the queue lengths at the nodes. So you can have a queue at the node. And what's gonna happen is you're gonna look at the difference in the queue lengths, that's called the back pressure. And then you're gonna flow depending on the back pressure. And in fact, this is gonna tell you how much to flow, like that. That's the way this is going to work. So here's a little baby example. You have five nodes and all the capacities are one here. And it's a single commodity network and I don't see where the input and output is. I believe the input's supposed to come in here and here, and it's supposed to go out there. It's interesting that it doesn't say it here. I guess it doesn't have to say it, does it? Because those are the s's. I guess that's fair. So here's the optimal flow. So these are given. That's in input. So stuff is pumped into the network here and it's pumped in here. And it's pulled out here. This, obviously, is the sum of these two.

So that's – call it a data collection network or whatever you want. By the way, we're minimizing the sum of the queuing delays on the network. And basically, that's actually the number of packets in the network, roughly. So that's what – sorry, no, I take that back. It's not that. It's the average queuing delay. Okay. So let's take a look at what happens. It says that what comes in here, I guess – I don't know why that doesn't – somehow, this should have the same width as that. I don't know why. That's very interesting. Wow. These are weird. It's spooky. Just ignore it. It's weird. It's obviously wrong. The only thing I'm concerned about here is the width of the – I guess the intention is that the width of the arrow shows you the flow level. But I think if you look at this, I'm getting some real bad – I'm getting kind of a KCL headache here. This does not look good. And this doesn't look good, because the sum of the width of this and this, I think, is probably that. And that goes here. So I'm not quite sure what we're seeing here. I'm going to have to think about this. We'll examine this and figure it out. Maybe it has an explanation. I'm just confused right now. Anyway, okay. So this is the optimal flow. And in particular, on these cross-lengths, there's zero flow. What's shown here, these are the potentials. For simplicity, I could add any number I like, like five, to all potentials and it would make no difference whatsoever. I just took the termination node and made that the zero or the ground potential. So these are the potentials here. And then you can look at the potential difference. Maybe that is the difference between that and that. Who knows? Can you eyeball them? It is. Maybe this tells you the potential difference, or something like that. The width of this is supposed to give you the flow level, but that's deeply suspicious, because for example, this thing is way too small or something here.

We'll have to figure out what actually happened here. Okay. That's the picture. And then here's how this would actually run if you did this. What you'd do is the following: this is just showing for several different step sizes here a constant. So here you just initialize all the potentials to one or whatever, and you get all sorts of flow errors, or something like that. And what happens is this, if you have a slow – if you take alpha is 0.3, this dual function shoots up and approaches what is the optimal one pretty quickly. One maybe goes up more quickly or something – maybe this is the point. Sorry, this is the 0.3. That's one and that's three. I got it all backwards. I should have just read the legend. So 0.3 is a nice, low step size. It's going to go up to the optimum. One is maybe the optimal step size, and three is so much that you're actually oscillating a little bit and you're not going to converge to the final, the actual solution here, you're going to converge to a neighbor of it or something like that. And 2.48 is going to be the optimal flow. Okay. Primal residual is going to be the norm of the flow error, and you can see things like see it go down and stuff like that with the different things. This one is not converging. It's going to converge merely to a neighborhood of the solution. And so, it's probably not going to get any better than that. That's the picture here. And here is the conversion of the potentials to their optimal values. I guess there's five nodes. One of them has potential zero, so we don't plot it. And this just shows how the other ones are going. And this is for the step size of one rule. And this shows the potentials going, converging to their optimal values. Okay. That finishes up these two applications. Let me just mention a couple things.

If you study these and get a feel for them, you'll be able to construct and solve problems that are much more complicated. And let me just mention a couple of examples. Let's do multi-commodity flow. So in multi-commodity flow, it's the same as a single-commodity flow problem, where you can have flows splitting and all sorts of other stuff, except you have different flows. They are completely separate. They are coupled by the costs on the edges, which could just be capacity costs, but they're coupled by the costs on the edges. Everybody see what I'm saying. So basically, you have the same trucks that take material from one place to another, but you have multiple products. And the number of trucks you use to send stuff from one warehouse to another doesn't matter. How much what the product makes in the truck is doesn't make any difference. It's the same price. So you have the multiple flows. That's one. You can go on and on. A common one in networks is that would convert both of these would be something like this: instead of having a fixed route from source to destination – that's our first problem – you would have ten good routes. So ten good routes. And you're allowed to split across those ten routes. So that's a common setting that's in between these two. Everybody see what that is? For each source destination pair, you have ten things. You can split the flows across them. So it's not like the single commodity flow, where you can completely split at every node the flows and then recombine them. This allows you to pump your traffic along ten different routes, or along any divisible combination of ten different routes. That's that one. Okay. So I think we'll quit here. And we'll start a new section of the class next time.

I have some questions for you. We haven't quite decided what we're going to do next. And let me tell you what some of the options are and maybe you have an opinion. They might have to do with your projects, but it's up to you. So one option is that we can jump on to these methods for solving extremely large problems, like 1 million variables and more. That's option one. It's very interesting material. That's one option. The other option is that we can do some applications that we've just worked on, like sequential convex programming, that's for non-convex problems. Is that a vote? We're getting a vote for that. Two. Okay. Three, four. How about solving the super large problems with truncated – wow, yikes. Don't worry, because they're not – it's only going to matter by a week. Wow. That was very close. You know what, I’m very sorry to say this, but we may have to wait until the convention to decide. Let’s try the hands one more time. Superdelegates for sequential convex programming please.

**Student:**Are there other options?

**Instructor (Stephen Boyd)**:Yeah, we got other options, but these seems to us are the ones that are going to be useful to you very soon. Sequential convex programming? It’s totally split. I’m not going to be reduced to counting. Okay, I will. How about the super large problems? Oh, God. That might have a slight edge. We will confer this weekend and decide in private in backrooms what’s going to happen. And then we’ll move forward.

[End of Audio]

Duration: 77 minutes