ConvexOptimizationII-Lecture04

Instructor (Stephen Boyd):Okay. I guess we're – we've started. Well, we get the video-induced drop-off in attendance. Oh, my first announcement is this afternoon at 12:50 – that's not this afternoon. I guess technically it is. Later today at 12:50, believe it or not, we're going to make history by having an actual tape-behind where we're going to go back and do a dramatic reenactment of the events that occurred at the first – on the first lecture. That's – so, I don't know, so people on the web can see it or something like that. That's 12:50 today. It's here. Obviously not all of you are going to come. But those who do come will get a gold star and extra help with their projects or who knows what. We'll figure something out. So please come because, although it's never happened to me in a tape-ahead or tape-behind, you know it's every professor's worst nightmare to go to a tape-ahead and have no one there. So, in fact, it's not even clear. Just some philosophical questions. And practical ones, like can you actually give a lecture if there was no one there? I pretty sure the answer is no. But okay. So we'll start in on continuing on the constrained subgradient method, projected subgradient method. Oh, let me make one more announcement. Homework 1 is due today, which has a couple things on it. We pipelined so homework two is currently in process. And then we're going to put out homework three later tonight, something like that. So, hey, just listen, it's harder to make those problems up than it is to do them. Come on. We can switch. You want to make up the problems and I'll do them? We can do that if you want. Fine with me. I can do – well, of course, your problems have to be well-posed and they have to actually kinda mostly be correct and kinda work. So anyway, we can switch if you want. Just let me know. Okay. So let's look at projected subgradient. So the projected subgradient method, let me just remind you what it is. It's really quite stupid. Here it is. It's amazing. Goes like this. You call f, f dot get subgrad. Get here at x to get a subgradient. So that's – you need a weak subgradient calculus method implemented. So you get a subgradient of f. You then take a step in the negative subgradient direction with a tradition step size. Of course, this in no way takes into account the constraint. And then you project onto the constraint. Now this is going to be useful, most useful, when this projected subgradient is meant to be most useful when this projection is easy to implement. And we talked about that last time. There are several cases where projections are easy. That's it. Projection on the unit simplex. That was it.

Homework 3 coming up. Coming up. Okay. Project on unit simplex. Okay. So obvious cases of projection on the non-negative orthant. Projection onto the cone of positive semidefinite matrices. But you'd be surprised. It's probably about 15, 20 sets that it's easy to project onto. Or easy in some sense. Of course, in – you can also project onto other sets, like for example, polyhedra or something like that. But that would involve using quadratic programming or something like that. Okay. This is projected subgradient method and a big use of it is applied to the dual problem. Now this is really a glimpse at a topic we're going to do later. So later in the class, we're going to look at this idea of distributed decentralized optimization. So far, kinda everything we've been talking about is centralized. You collect all the data in one place and calculate gradients and all this kind of stuff. In fact, we're going to see beautiful decentralized methods, and they're going to be based on this. So this is a glimpse into the future. Maybe not even too far a

future. Maybe like a couple of lectures or something. But let's see. Okay. So we have a primal problem, minimized f0 subject to fi less than zero. And we form this dual problem, which is maximized the dual function at lambda. These are the Lagrange multipliers lambda. These have to be non-negative because these are inequality constraints like this. And the projected subgradient method is very easy, because we're maximizing this concave function subject to the constraint that you're in the non-negative orthant. Projection on the non-negative orthant is completely trivial. You just take the plus of it – of the vector – component by component. So the update's going to look like this. You will find a subgradient of minus g here. So we'll find a subgradient of minus g, and then we will step in the negative subgradient direction. Actually, I suspect this is correct, but this could be a plus here. I don't know. And the rule is for 300 level classes, I don't even care. If it's a plus then you fix it or something like that. I actually think this is right. It's confusing because we have that subgradient of minus g. Okay. So you take a subgradient step in the appropriate direction, so I'm allowed to say that in a 300 level class, and then you project here. So that's it. Now by the way, I should mention again kinda going towards – when – if I solve this dual, what are the – when is it that I can actually extract the solution of the primal problem? Again, this is 364a material, which we covered way too fast, but I don't know. Does anyone remember? Anyone remember this?

Student:[Inaudible].

Instructor (Stephen Boyd):Sure, we're going to need strong duality holds if it were strictly feasible. We'd have Slater's condition and strong duality would hold. That gives you zero duality gap and I guess if you don't have that, then you can't solve this at all, because the optimal values aren't even the same. So let's assume that. There's more, actually, to it than just that. What the sledgehammer condition is is this. What you'll need is that when you find lambda*, what you want is that the Lagrangian at lambda* should have a unique minimizer in x. If it does, then that x is actually x* up here. Okay? So that's the condition. A simple condition for that – you should go back and read this, because we're going to be doing this in a couple of weeks and actually these things are really going to matter. So we're going to do network flow methods and all sorts of other stuff, and they're actually gonna matter. Here's the – one sledgehammer condition is f0 here is strictly convex. Because if f0 is strictly convex then f0 plus lambdai fi, where fi are convex, is also strictly convex. For all lambda, including all lambda0 it doesn't matter. It's strictly convex. If it's strictly convex, it has a unique minimizer. So just to go back to this, you would actually calculate the optimal lambda*. You would then go back and get the minimizer – the minimizer of the Lagrangian with respect over x, call that x*. And that will actually be the optimum of this problem. Okay. So let's work out the subgradient of the negative dual function. So it's actually quite cool. Let's let x* of lambda be this. It's arg min of the Lagrangian. So it's just exactly what we were just talking about. And here's sorta the big sledgehammer assumption is f0 is strictly convex. And by the way, in this case, you might say a lot of times some of these things are silly. They're sorta things that basically only a theorist would worry about. I mean, somebody should worry about them, but they have no implications in practice. And I'm very sorry to report that this is not one of those. This actually – there are many cases in practice with

real methods where this issue comes up. It's real. It means that methods will or will not converge and you have to take extra effort in things like that. Okay. All right, so we'll just make this sledgehammer assumption here, the crude assumption, f0 is strictly convex. That means this is strictly convex here. And therefore, it has a unique minimizer. And we're going to call that minimizer x* of lambda. It's a function of lambda. Okay? So that's x* of lambda.

And of course, if x* is the minimizer, than g of lambda is f0 of x* of lambda plus lambda1 times this. It's the Lagrangian evaluated lambda*. Okay. So a subgradient of minus g at lambda is then given by this. It's hi is minus fi of x* of lambda. Now this is actually quite an interesting – first of all, let me explain that. Let me see if I can get this right. g is the infenum over z – it's the infenum over z of this Lagrangian here. That's the infenum of – that's what g of lambda is. So negative g of lambda is a supremum. How do you calculate the subgradient of a supremum? No problem. You pick a point that maximizes – one of the points that maximizes. In this case there's a unique one. That's what this assumption says here. So you pick the maximizer, that's this, and then you form this thing, and then you ask yourself what is the gradient, subgradient, of this thing with respect to lambda? That's an affline function. So the subgradient is simply this thing here up to this thing here. And so, again, modular minus sines. My guess is that this one's correct, but I guess we'll hear if they're not. And we'll silently update it. But I think it's actually right. So a subgradient of minus g is this. By the way, that's a very interesting thing. Let me say what that is. This – if this is positive, then let's see, what does that mean? If hi is positive – maybe we don't – well, we can work it out. If fi is negative, that means that the ith inequality constraint is satisfied. If it's positive it means it's violated. So that means that hi if it's positive is something like a slack. So hi is a slack in the ith inequality. If hi is positive it means the ith inequality is satisfied. If hi it is violated. And hi is the amount by which it's violated. Okay. So here's the algorithm. Here's the projected subgradient method, just translated using the subgradient. Notice how embarrassingly simple it is. So this is projected subgradient method for the dual. And it basically says this. It says you start with some lambdas. You can start with all lambdas equal to one. For that matter, start with all lambdas equal to zero. It doesn't – just start with all lambdas zero. It says at your current lambda, minimizes Lagrangian without any consideration of feasibility for the primal problem. Now when you minimize this thing here, and basically the lambda are, of course, prices or costs associated with the constraints. So this is sorta a net here, because it's sorta your cost plus, and then these are charges and subsidies for violations. It's a charge if it's a violation. And it is a subsidy if fi is negative, which means you have slack. And then, actually, you derive income from it. Okay. So that's the meaning of this. So it says what you do is you set all – it's basically a price update algorithm and you start with any prices you like. They have to be non-negative. Start with the prices. You then calculate the optimal x. No reason to believe this optimal x is feasible. By the way, if the optimal x is feasible, you're done. You're globally optimal. So if fi of x* at any step, if they're all less than or equal to zero, you're done, optimal. And not only that, you have a primal dual pair proving it.

Okay. Otherwise what you do is you do this. And this is really cool. You go over here and you look at fi of x. If fi of x, let's say, is plus one, it means that your current x is

violating constraint i. Okay? It says you're violating constraint i. That says you're not being – the price is not high enough. So it says increase the charge on resource one. If – resource i if fi represents a resource usage. So it says pop the price up in that case and alpha tells you how much to pop the price up. In that case, the plus is totally irrelevant, because that was non-negative. You adding something. You bumped the price up and there's no way this could ever come into play. Okay. So it says – I mean, this is actually – this is the name – I should say the comment, if you make a little comment symbol over here in the code you should write on the thing "price update." Because that's exactly what this is, the price update. So what you do then is this. If fi is negative, that's interesting. That means you're underutilizing resource i. It's possible that in the final solution the constraint i is not tight, in which case it doesn't matter. That's fine. That's the right thing to do, but you're underutilizing it. And what this says is in that case, that's negative, this says drop the price on that resource. This says drop the price. However, now this comes into play. It says drop the price, but if the new price goes under zero, which messes everything up, because now it encourages you to violate inequality, not satisfy them, then you just make the price zero. And so, for example, if the ith inequality is, in the end, gonna be at the optimal point, this is actually gonna be not tight, then what's going to happen is that price is gonna go to zero like that. You're gonna – at the – you'll be underutilized here. That'll be negative. That'll be zero for the last step. This will become negative. The plus part will restore it to zero. So this algorithm, I mean, it's actually a beautiful algorithm. It goes to the – variation on this go back into the '50s and '60s, and so – and you find them in economics. So this is – it's just a price update or, I think, this is one – this would be part of a – a bigger family of things. I guess they call this a TETMA process, or something like that, where – I don't know. Who's taken economics and knows the name for these things? Does anyone remember? Come on, someone here took an economics class and saw some price adjustment. Okay, let's just move on. No problem.

All right. So in that method, it says that the primal iterates are not feasible. That's, I mean, it's actually – if you ever hit an interation where the primal iterates are feasible you are now primal dual optimal, quit. You quit with perfect – so what it means is in a method like this, projected subgradient applied to the base problem, after each step you're feasible because you projected onto the feasible set. So that's a feasible method. And all that's happening is your function value is going down. I might add not monotonically, right. So these are not decent methods. But your function value's coming down the optimal one non-monotonically. In a dual subgradient method what's happening is that the primal iterates are not feasible. What happens is these things – you're approaching feasibility. That's what happens. In fact, you'll never hit feasibility. If you hit feasibility you terminate at the end. And in this case, the dual function values, all of which are lower-bound – so the one nice part about this dual subgradient method is at each step you have a global lower bound on the original problem. You do add value at g exactly at each step. So you have a lower bound. By the way, there's a couple of tricks. I think these are in the notes, so if you read the notes, this becomes especially cool when you have a way, some special method for taking x tilde, and from it, instructing – I want to say projection, but it doesn't have to be projection – but constructing a feasible point. Could be by projection. So you can get – you can construct a feasible point, then this

algorithm will actually produce at each step two things, a lower bound, a dual feasible point. You'll know g of lambda. That's the lower bound of your problem. It will go up non-monotonically. And you'll also have a feasible point called at x tilde of k. Tilde is the operation of constructing a feasible point from x(k). And then you get primal points whose function value is going down non-monotonically. Then you actually get a duality gap and all that kind of stuff. Okay. So I think we've talked about all this. That's the interpretation of the thing. It's really quite beautiful. The coolest part about it you haven't seen yet. And the coolest part we're going to see later in the class. Not much later, but it's that this is going to yield a decentralized algorithm.

For example, you can do network flow control, you can do all sorts of crazy stuff with this, and it's quite cool. But that's later. Okay. So we'll just do an example just to see how this method works, or that it works, or whatever. Oh, I should mention something here. And you might want to think about when would this algorithm be a good algorithm. When would it look attractive? And let me show you, actually, one case just right now immediately. This is trivial calculation. The only – the actual only work is here. And what this means is you have to do, at each step, the work is actually in minimizing this Lagrangian. So basically, at each step there will be prices, and then you minimize the Lagrangian. That's going to be the work. Therefore, if at any time you have an efficient method for minimizing this function, you are up and running. Okay? So, for example, I mean, I can name a couple of thing. Someplace you have a control problem and these are quadratic functions. Then if you have your – if this weighted sum is also going to be one of these convex control problems that it means you can apply your LQR or your Riccati recursion or whatever you want to call it to that. If this is image processing and somehow this involves something involving 2D df keys and all sorts of other stuff, the grad student before you has spent an entire dissertation writing a super-fancy, multigrid blah, blah, blah that solves this thing well. If it solves least squares problems there, and if this is a least squares problem, you're up and running. And you wrap another loop around that where you just update weights and then repeatedly solve this problem. So it's good to be on the lookout. Any time where you see – where you know a problem where you have an efficient method for solving – actually, just a way of minimizing a weighted sum – this is what you want to do.

Okay. Let's look at an example. This is going to be quadratic minimization. It's not a big deal. And we'll make p strictly positive over unit box. So notice here we can do all sorts of things with this. Oh, we could do the projected subgradient primal is easy here. So projected subgradient primal goes like this: you take x at each step and then you x minus equals alpha times p(x) minus q. Everybody follow that? That was x minus equals alpha g. And then I take that quantity and I apply sat, saturation, because saturation is how you project onto the unit box. Everybody got that? That's the method. Okay? So okay, that would be primal subgradient method applied to this. By the way, I should mention something here, that is, don't – these are not endorsements of these methods, and in fact, these methods only make sense to actually use in very special circumstances. If you just want to solve a box-constrained QP like this, and x is only 2,000 dimensions or it could be all sorts of other things, you are way better off using all the methods of 364a. That's just an interior point method. So actually, if someone said, "Oh, I'm using primal

decomposition or dual decomposition to solve this," I would actually really need to understand why. There are some good reasons. One of them is not, I don't know, just because it's cool or something. I mean, that's not – I mean, here, for example, this would be so fast if you made a primal barrier method for it. It would be insane. So there are only special reasons why you'd want to do this. One would be that when you write down this dual subgradient method it turns out to be centralized. That would be – that works as a compelling argument. But just to remind you, these methods are slow. They might be two lines, like that. I guess if you put a semicolon here, it's one line. They might be two lines. They might be simple. But they're not – these are not the recommended – I just want to make that clear. Okay. So here's the Lagrangian. And, indeed, it is positive – this is positive definite quadratic function for each value of lambda, because you don't even need this part. It's positive definite already here. And so here's x*. It's this. It's p plus 9 ag of 2 lambda inverse q. And the projected subgradient method for the dual, this looks like that. So you, in fact, it makes perfect sense. It even goes really back to 263 and it goes back to regularization. If you didn't know anything about convex optimization, but you knew about least squares, that describes a lot of people, by the way, who do stuff.

And by the way, people who do stuff and actually get stuff done and working. So don't make fun of them. Don't ever make fun of those people. So how would a person handle that if you hadn't taken 364? Well, it'd be 263. You'd look at it and you'd say, "Well, I know how to minimize that." That's no problem. That's that, without the lambda there. That's p inverse q. Something like that. And then you'd look at it and you'd go, "Yeah, but, I mean, this is a problem." So here's how a human being would do it. They'd do this. They calculate p inverse q. That's x. If that x is inside the unit box, they would say – they'd have the sense to say, "I'm done." Otherwise, they'd say, "Oh, x7 is like way big. Ouch. That's no good." So I will add to this. I will regularize and I will put plus a number, some number, times x7 squared. Everybody cool on that? You're adding a penalty for making x7 big. Okay? And you'd look at this and be like x12 is also big and you'd add something there. I'm not – remember, don't make fun of these people. I don't. You shouldn't. So then you'd solve it again. And now – except it would be smaller. Now x7 is too small. Now x7 has turned out to be plus/minus – is 0.8. And you go, oh sorry, my weight was too big. So you back off on x7 and now other things are coming up and over. And you adjust these weights until you get tired and you announce, "That's good enough." Okay. I mean, listen, don't laugh. This is exactly how engineering is done. Least squares with weight twiddling. Period. That's how it's done. Like if you're in some field like machine learning or something you think, oh now, how unsophisticated. People in my field are much more sophisticated. This is false. All fields do this. This is the way it really works in those little cubicles down in Santa Clara. This is the way it's done. You're doing imaging. You don't like it. Too smoothed out.

You go back and you tweak a parameter and you do it again. So no shame. All right. So, actually, if you think about what this method is, this is weight twiddling. That's what this says. It's weight twiddling. It says pick some regularization weights, because that's what these are, and then it says update the regularization weights this way in a very organized way. It just – you just update them this way. So this is, in fact, a weight twiddling – an economist would call this a price update algorithm. And maybe an engineer might call it

a weight twiddling algorithm. They might even – there's probably people who invented this and didn't know it. Anyway, everyone see what I'm saying here? Okay. Let me ask you a couple of questions about it, just for fun, because I've noticed that the 364a material has soaked in a bit. If p – not fully. If p is banded, how fast can you do this if p is banded? Let's say it's got a bandwidth around k. N is the size of x. How fast can you do that?

Student:[Inaudible].

Instructor (Stephen Boyd):With n squared k? That's your opening bid? That's better than n cubed, right. If p is full, that's n3. That's a Cholesky factorization and a forward and backward substitution, right? Let's make p banded. You said n squared k, that was your opening?

Student:[Inaudible].

Instructor (Stephen Boyd):Oh, even better. So nk squared. You're right. That's the answer. Okay. So just to make it – I mean, you want me to – let me just make a point here. If this is full, you probably don't want to do this for more than a couple of thousand. Three thousand, 4,000, you start getting swapping and stuff like that on something like that. You have a bunch of machines, all your friends' machines, and you run MPI and all that stuff. Whatever. You can go up to 5,000, 10,000, something like that. But things are getting pretty hairy. And they're getting pretty serious at that point. If this thing is blocked – if p is block-banded or something like that, it's got a bandwidth of ten, how big do you think I could go? For example, my laptop, and solve that. Remember, the limit would be 1,000. I could do 2,000. It's growing like the cube, so every time you double it goes up by a factor of ten or eight or whatever. So what's a rough number? Well, put it this way, we wouldn't have to worry at all about my laptop about a million. I want to make a point here that knowing all this stuff about structure and recognizing the context of problems puts you in a very, very good position. By the way, where would banded structure come up in a least squares problem? Does it ever come up?

Student:[Inaudible].

Instructor (Stephen Boyd):Structures that are, yeah, that actually banded – what does banded mean? Banded means that x(i) only interacts with x(j) for some bound on i minus j. So if you had a sort of a trust or some mechanical thing that went like this and things never – bars never went too far from one to the other, that would be a perfect example. Let me give you some others. Control, dynamic system. So just control is one, because there, it's time. And, for example, if you have a linear dynamical system or something like that, the third state at time 12, it interacts, roughly, with states one step before and one after. But then that's banded. How about this? How about all of signal processing? There's a small example for you. All of signal processing works that way, more or less. Because there the band structure comes from time. Signal processing means that each x is dependent only on a few – you know, a bounded memory for how much it matters. Now the whole problem is coupled, right? Okay. This is just for fun, but I'm going to use – it's

good to go over this stuff, because, okay, I just use it as an excuse to go over that. Okay. So here's a problem instance. So here's a problem instance where I guess we have 50 dimensions and took a step at point one. Oh, I should – I can ask a question here about this. In this case, it turns out g is actually differentiable. So if g is differentiable, that actually justifies theoretically using a fixed step size. Actually, in practice as well, because in a – if you have a differentiable function, if you apply a fixed step size, and the step size is small enough, then you will converge to the true solution. So this goes g of lambda. These are lower bounds on the optimal value, like that. They converge. And this is the upper bound, found by finding a nearby feasible point. And then let me just ask you – I don't even know because I didn't look at the codes this morning on how I did this, but why don't you guess it?

At each step of this algorithm, here, when you calculate this thing – by the way, if this thing is inside the unit box, you quit and you're done. You're globally optimal because you're both primal and dual. End of story. Zero duality gap. Everything's done. So at each step at least one of these – at least one component of this pops outside the unit box. Please give me some guesses – give me just a uristic for taking an x and producing from it something that's feasible for this problem.

Student:[Inaudible].

Instructor (Stephen Boyd):Simple? What do you do?

Student: If the [inaudible] is negative one, you make it one. If it's less than negative one, you make it negative one.

Instructor (Stephen Boyd):There you go. You just project. So in this case it's too easy to calculate the projection. You just calculate the projection and so, in fact, this – whoops. This thing here – I'm sure that's what this is, but x tilde is simply the projection of x(k) onto the unit box, so that's what that is. Okay, so that's that. We're going to come back and see a lot more about projected subgradient methods applied to the dual later in the class. Okay. So let's look at a more general case that's going to be subgradient method for constrained optimization. So here, instead of describing the constraint as just a constraint set, we'll write it out explicitly as some convex inequalities. So this goes like this. Here's the update. I mean it's really dumb. I'll tell you what it is. You simply do a subgradient step. And here's what you do. If the point is feasible, you do an objective subgradient step. If it's not feasible, then you find any violated constraint and you use a subgradient of that. Okay? Does this make sense? So it's really quite strange. In fact, what's kinda wild about it is that it actually – I mean, that it actually works. So, I mean, you realize how myopic this is. It's very, very silly. It basically – so the algorithm goes like this: you're given x and you start walking through the list of constraints. So you evaluate f1, f2, f3. If those are less than or equal to zero, you go to the next one. The first one – I mean, that's just a valid method. The first time you hit a violated constraint, that j is positive, fj, you simply call fj dot get subgrad, or something like that, to generate a g, and you take a step in that direction. Does that reduce fj? No. Subgradient method is not a decent method. There's no reason – so basically you go down, you find the 14th

inequality. If violated, you take a subgradient step, and that could, and often does, make the violation of the 14th inequality worse. I mean, the whole thing is like – these algorithms are just highly implausible.

I mean, the kinda things where you need the proof because the algorithms themselves are so ludicrous. Okay. Now here we have to change a few things. fk best is the best objective value we have over all the points that are feasible, and this can actually be plus infinity if you haven't found any feasible points yet. So fk best is initialized as plus infinity. Okay, so the convergence is basically the same. I won't go into the details. It just works. I mean, that's the power of these kinda very slow, very crude methods. In fact, that's going to come up in our next topic. What are you going to say about a subgradient method is they're very unsophisticated, they're very slow, but actually, one of the things you get in return for that is that they are very rugged. In fact, in the next lecture, which we'll get to very soon, you'll see exactly how rugged they are. There it kinda makes sense. Anyway, so there it is. That's a typical result. And I think the proof of this is in the notes. You can look at it. But let's just do an inequality form LP, so let's minimize C transpose x subject to Ax less than b. It's a problem with 20 variables and 200 inequalities. Let's see, the optimal value for that instance turns out to be minus 3.4. We have one over k step size here. Oh, by the way, when we do the feasible step, you can do a Polyak step size, because when you're – if you're doing a step on fj, which is a violated inequality, what you're interested in is fj equals zero. You are specifically interested in that. So your step size can be the Polyak. And this would be an example of sorta the convergence f minus f*. I guess if f* is minus 3.4, then – well, this is not bad, right? I mean, this is – I don't know. Let's find out where 10 percent is. There's 10 percent.

So took you about 500 steps to get 10 percent or something. So each step here costs what? What's the cost of the step here, assuming dense, no structural? What's the cost of a step in solving? Let's write that down. So we're gonna – let's see – we're gonna solve this problem. We're gonna minimize C transpose x subject to Ax less than b. What's the cost of a subgradient method step here? You're exempted because you can't see it. Is that true you can't see it? No, you can't see it. You can see the constraints. That's actually the really important part. What's the cost?

Student:[Inaudible].

Instructor (Stephen Boyd):How did you do it? What's the method? If you were in matlab, how long would it be? For that matter, it's just as easy to write it in lapad, but let's write it in matlab. What would it be? How do you implement this method here? Not this one, but – by the way, of course all the source code for this is online, but there's a method, right? So what's the method? Somebody tell me the method. Homework three. You'll be doing this shortly enough. Well, here's the lazy way. You just evaluate Ax and you compare to b. If Ax is less than or equal to b, what's your update on x? It's x minus equals alpha c, right? Otherwise, if Ax is not less than or equal to b, you sorta, you find – for example, you might as well find the maximum violated one, or – I mean, it doesn't matter. That's the point. You can take any violated one. So but if you evaluate all of them

– and that's just from laziness – you evaluate all of them, then what's the cost, actually, of evaluating this? There's your cost right there. It's just multiplying Ax. What's that?

Student:[Inaudible].

Instructor (Stephen Boyd):Are you saying mn? Thank you, good. Okay, mn. This is – it's irritating – I know – the thing is, you should know these things. This should not be abstract parts from three days of 364a. You should just know these things. You should know what the numbers are on modern processors and things like that. Just for fun everyone should – after a while, then, we quit and then you go back and it's just Ax and stuff like that. So the cost is mn per step. So what that says – whereas, how about – what's the cost on an interior point method on this guy? What's an interior point step? In fact, what's the interior point method complexity period, end of story, on this guy? Just minimize C transpose Ax less than b. At each step you have to solve something that looks like Ad, A transpose, something or other, right? And that's going to be n cubed – I mean, unless that's [inaudible], that's n cubed. But forming Ad A transpose, that's the joke on you. That's m(n)2. m is bigger than n. That's the dominant term. So it's m(n)2. How many interior point steps does it take to solve this problem?

Student:[Inaudible].

Instructor (Stephen Boyd):Thank you. Twenty. So it's 20. So what's the overall complexity of solving this problem? m n squared. And remember what that is that follows my mnemonic. It's the big dimension times little dimension squared.

This assumes you know what you're doing. If you do it the wrong way, it's the big dimension squared times the small dimension. Always ask yourself that. So it's m n squared versus mn. So basically, a subgradient step costs a factor of n more. n is 20. I mean, it doesn't – is that right? Okay, so that says here you really should divide these by 20. And so that – so you said 20 steps. So this is 25. It would – you would actually have solved the problem here. You've solved it through about 10 percent accuracy with this subgradient type method here, roughly ten percent. Maybe a little bit better. But in an interior point method, in this amount of effort, roughly, in this amount of effort you'd have the accuracy of ten to the minus ten. Okay. Everything cool? All right. Is that going to work? I don't know what I've done. Okay. So, our next topic is the stochastic subgradient method. And we're gonna get to some of the first things we can actually do with subgradient-type methods that don't really have an analog in interior point methods. We're not – so far they're just cool because they're three lines that proof of convergences, four lines and so on. We're gonna see now some very cool things about subgradient methods later, but now we're gonna see something that's actually different and isn't 364a compatible. So we're gonna do stochastic subgradient method. And let me just give you the rough background on it.

The rough background on it is that these methods, these subgradient methods, they're slow, but they're completely robust. They just don't break down. They're three lines of code, one really, two, something like that. One or two lines of code, two lines of code.

They're very slow and boy are they robust. They just cannot be messed up. And we're gonna see a very specific example of that. In fact, what's gonna turn out is you can add noise, not small noise, to the subgradient calculator. Everyone would guess – and look, if you're doing stuff in double precision floating points, you're always adding noise every time you all anything. In an interior point method you get – you say get me the gradient. That comes back with noise. I guess in ee, we would say with something like 200 decibel signal noise ratio, because that's what a i triple-e floating point gives you. But it basically comes back already with noise, but it's on the order of 1(e) minus 8 or 1(e) minus 10 times the size of the thing you're calculating. That's standard. It's gonna turn out there. So no one would be surprised if barrier method continued to work if there was noise that was in the sixth figure of your gradient calculation. That would hardly be surprising. Fifth figure, again. Fourth figure you can start imaging having some trouble now. The subgradient methods, they work this way. Here's how stupid and robust they are. Not only – you can actually have a signal of noise ratio that's quite negative. So notice you can have basically a subgradient where the signal noise ratio is one. In other words, that means basically when the person says the subgradient is that direction, the true subgradient can actually be back there. It's just sorta if you ask them 50 time or 100 times or something, they should be kinda average out vaguely to the right direction. Everybody got this?

So it went to school, actually. It has lots of applications. So okay, so let me define a noisy unbiased subgradient. So here's what it is. So I have a fixed – this is a deterministic point, x, and then I have a noisy unbiased subgradient for f at x is this. It is a vector, a random vector g tilde that satisfies this, that its, on average, its expected value is a subgradient. Now by the way, this means, of course, that for any particular g tilde, this inequality if false, I mean, obviously, need not hold, right? However, on average – so basically think of your f dot get subgrad as being ran – it's not deterministic. When you call it, it gives you different g's. If you call it a zillion times on average that would give you something close to the mean. That's close to a subgradient. Everybody got it? So we'll see lots of practical examples where you get things like that. Okay. Another way to say it is this, is that what comes back is a true subgradient plus a noise, which is zero mean. That's a stochastic subgradient. Now this error here, it can represent all sort of things. It can be – first of all, it can just be computation error that basically when you calculate subgradient you're sloppy or you do it in pix point or I don't know. Anything like that. But it can also be measurement noise.

We're going to see it's going to be Monte Carlo sampling error, so if, in fact, the function itself is an expected value of something, and you estimate an expected value by Monte Carlo, then you're right, it's unbiased – I mean, it's an unbiased estimated. You write it down as – well, it's an unexpected then the average is the right. But you get – it's unbiased, and then v is actually the difference between – it's a random variable and it's the difference between the what you actually get is your Monte Carlo sampling error. Okay. Now if x is also random, then you say the g tilde is a noisy unbiased subgradient if the following is true: for all z this holds almost surely that this is the conditional expectation of g tilde. That's the noisy subgradient condition on x. Now that's a random variable, so this right-hand side is a random variable. That's not a random variable. And

it's also a random variable because x is a random variable. So the whole thing on the right is a random variable. And if this inequality holds almost surely, then you call it a noisy unbiased subgradient. So that's what it is.

Okay. And that's the same as saying the following: it says that the conditional expectation of g tilde given x is a subgradient of f at x almost surely. So that's what it means. For the conditional one, if x is not random, it's like that, I can – I don't need the condition on x and I can erase that. So, let's see, I don't know what this means. Anyway. This is a random vector. That's a random vector and the idea is – and that's actually a random set. And so it basically says that that inequality holds almost surely. So okay. Now here's a stochastic subgradient method. Ready? Here. In other words, it's the subgradient method. So it says you got a noisy subgradient, I'll just use it. I'll just use it. Nothing else. You basically update like that and that's it. Now I want to point something out. You get a – this is now a stochastic process, because even if x(0), if your initial x was deterministic, then g(0) is already a random variable, and therefore, x(1) is a random variable, the first update, because it depends on g(0). And so this is now a stochastic process, this thing. Okay. So we now have the stochastic process, which is the stochastic subgradient – the trajectory of the stochastic subgradient method. And here you just have any noisy unbiased subgradient. And then we'll take the step size, the same as always, and then f(k) best is going to be the min of these things. That's a – by the way, that's a random variable there. Because that's now a stochastic process here. So that's a stochastic process and that's a random variable. It is f(k) best. Okay. So here's some assumptions. The first is we'll assume that the problem is bounded below. These are much stronger than you need, but that's good enough. We'll make this global Lipschitz condition here. More sophisticated methods you can relax these, but that's okay. And we'll take the expected value of x(1) minus x* – x(1), by the way, could be just a fixed number, in which case you don't even need this expected value.

It's the same as before. Now we're going to have the step sizes, they're going to be square-summable, but not summable. So, for example, one over k would do the trick. So you're going to take a l2, but not l1. Little l2, but not little l1 sequence of steps. One over k is fine. Okay. Here are the convergence results. Okay, I'll summarize this one. It works. This says that the – that it converges in probability. And, in fact, you have almost sure convergence. We're not going to prove this one, although it's not that hard to do, this one. We will show – actually, we'll show this. This will follow immediately from that, since these are f(k) minus – f(k) is bigger than f*. So that'll follow immediately. So before we go on and look at all this, I just want to point out how ridiculous this is. So first of all, the subgradient method by itself I think is ridiculous enough. It basically says you want to minimize – you want to do minimax problem. It says no problem. At each step go around and find out which of the functions is the maximum. If there's more than one, arbitrarily break ties. Return, let's say gradient of that one, and take a step in that direction. That's totally stupid, because if you're doing minimax, the whole point is when you finish, a lot of these things are going to be tied, and the whole point is you don't want to just step greedily to improve one of these functions when you have a bunch of them. It just says do it, and the one over k step size are going to take care of everything. What's wild about this is that that method, though, is so robust that, in fact, your get subgradient

algorithm can be so bad that it can actually, as long as, on average, it's returning valid subgradients, it's gonna work. So single to noise ratio can be minus 20 decibels. You can be getting the – whenever you get a subgradient, you could be adding to that a noise ten times bigger than the actual subgradient. Everybody see this? The whole thing is completely ridiculous.

Now, how – will the convergence be fast? No. It can't be. I mean, it can hardly be fast if someone's only giving you a subgradient, which is kind of a crappy direction anyway for where to go. But now if they give you a subgradient where the negative 20 decibels signal noise ratio, in other words with – basically it says that you can't even trust the subgradient within a factor of ten. You'd have to call – you'd actually ask for a subgradient direction like ten or 100 – you'd call it 100 times and average the answers. And that's the only time you could start getting some moderately sensible direction to go in. Everybody see what I'm saying here? The whole thing's quite ridiculous. And the summary is, it just works. These are kinda cool. What? It's also – this is known and used in a lot of different things, signal processing and all sorts of other areas. Actually, there's a big resurgence of interest in this right now in what people call online algorithm that's being done by people in CS and machine learning and stuff like that. So let's look at the convergence groove. It's tricky. You won't get the subtleties here. But you can look at the notes, too. It's not simple. I don't know. I got very deeply confused and you have to go over it very carefully. That it's subtle you won't get from this, but let's look at it. It goes like this. You're going to look at the conditional expectation of the distance to an optimal point given x(k) – the next distance here. Now this thing is nothing but that, so we just plug that in. And we do the same thing we did with the subgradient method. We split it out and take this minus this. That's one term. And we get this term. Now that, and this is conditioned on x(k). So x(k) conditioned on x(k) is x(k). So this loses the conditional expectation condition on x(k). That's a random variable, of course. But you lose the conditional expectation. It's the same.

Then you get this. You get two alpha times the conditional expectation of now it's the cross-product of this minus this and that term. And that's this thing conditioned on x(k). And the last term is you get alpha squared times the conditional expectation of the subgradient squared given x(k). And we're just going to leave that term and leave it alone. Now this term in here, we're going to break up into two things. We're going to write it this way. It's the – I can take here the x* is a constant and so condition on x(k), that just x*. And then this term, g tilde k transposed x*, that's linear in this, so conditional expectation commutes with linear operators, so that comes around and you get this thing. Now that – this thing here, the definition of being a subgradient noisy stochastic subgradient, or a stochastic subgradient, if you like, is that this thing here should be, I guess it's bigger than or equal to whatever the correct inequality is this, to make this thing true. So that's how that works. And so you end up with this. Now if you go back and look at the proof of the gradient method, subgradient method, it looks the same, except there's not the conditional expectations around. And there's a few extra lines in here because of the conditional expectation. So let's look at this. And this inequality here is going to hold almost surely. Everything here is a random variable. That's a random variable. This entire thing over here is a random variable. So this

inequality holds almost sure this thing is less than that. And now what you can do is the following: we can actually telescope – I mean, we can actually now telescope stuff, the same as before. If we take – I should say, if we take expectation of this now, then the expectation of this is just the same as the expected value of that. That's a number, and that's less than the expected value of that minus, then, the expected value of that.

Expected value of that, that just drops the conditional part there. And so here's what you get. You end up, if you take expectation of left- and right-hand sides of the inequality above, which was an inequality that holds almost surely, you get this. The expected distance to the optimal point in the next step is less than the expected – the current distance to the next point minus two alpha k times the expected value of your suboptimality here, plus alpha squared times the expected value of the subgradient squared. This thing will replace with just the number g squared here, and we'll apply this recursively and you end up with this, that the expected value of x(k) plus 1 squared minus x* is less than the expected value of x(1) minus x squared. This is going to be less than r squared here, this thing. That's our assumption. And then again we get the good guy and the bad guy. That's bad. This is good because this is – I mean, this thing is always bigger than that by definition. So whatever this is here, it's a number. Whatever this number is, it's non-negative here. I guess it's a positive, or something like that. That's a positive number. That's a negative sign here, so this is on our side. It's actually making this distance smaller. And the nice part about that is that goes in alpha, this goes in alpha squared, and so the bad guy, at least for small enough alpha, is gonna lose. And then you just simply turn this around and you get the following: you get the minimum of i equals 1 to k of the expected value of your suboptimality, which is less than or equal to r squared plus g squared and norm alpha squared. It's the same thing as before. So it's exactly what we had before. Okay. Except now it's the minimum of the expected value of the suboptimality. That's actually a random variable here. So that's the difference.

Okay. Now this tells us immediately that the expected value of – that this sequence actually converges the min of these expected values converges to f*. That's what it tells us. Actually, I believe that the fact is, you don't even need the min here. This converges. That's a stronger result, but we don't need it. We'll get to that. Now we're going to apply Jensen's inequality. So Jensen's inequality says the following: this is – I'm gonna commute expectation and min. Min is a concave function, so what that says is the following: Here I have the expectation of the min and here I have the min of the expectation. And I guess the inequality goes this way. But this thing here is a random variable and it's a random variable f(k) best. And so expected value of f(k) best is less than or equal to this. This thing goes to zero. So we're done. By the way, I never remember which way Jensen's inequality goes. So I'm not ashamed to admit it. So I usually have to go to a quiet place or draw a little picture with this and some lines. Because every time you do it it's something different. It's either a concave or whatever. It's important which way it goes, so I just go somewhere quiet and I draw a picture and then come back and see. I'm trusting myself that it's correct. I think it is.

But once you know this, you're done. But this is – now you're done with – you can get convergence of probability very easily, because these random variables are non-negative.

So these are non-negative. So the probability that a positive random variable's bigger than epsilon is less than that, and we already know the numerator goes to zero, so for any epsilon, this goes to zero. And so you get convergence in probability. So by the way, this is not – this is not simple stuff. I mean, it's not complicated. It did fit on two slides with giant font size. But trust me, it's not – it's not totally straightforward. So and I think the notes has this in more detail. Okay. Let's do an example. So here's an example. Least lines linear minimization. This is what we're going to do. We're going to use the stochastic subgradient method, except that – how do you get a subgradient of this thing? How do you get a subgradient of this? What do you do? Yeah, you evaluate like all – you have to evaluate all of these. Because otherwise you don't know what the maximum is. You evaluate all of them, find the maximum value, then go back and find one of them that had maximum value. Break ties arbitrarily. Could be the last one that had maximum value. And then you return ai. So here's what we're going to do. We will actually artificially we'll just add zero mean random noise to it. We'll just add noise. So we'll make, basically we'll make – we'll add noise in our f dot get subgrad method. That's what we're gonna do. And here's just a problem instance with 20 variables. M equals 100 – you have 100 terms. You've seen this before. f* is – the same example. One over k step size. And the noises are – have about a four – they have about 25 percent of the sizes of the subgradient, because the average size of the subgradient is about four and the noise is the average size is – let's see, each of these is about .7 or something like that. So that's about 25 percent, something like that.

Okay. And here's what happens. If you have – here's the noise-free case. So you get this. And so, indeed, so I should say what this means is you're getting the subgradient at about how many bits of accuracy are you getting in your subgradient? When you call subgradient, how many bits of accuracy are you getting here? I mean, if your noise is on the order of a quarter, I just want a rough number. How many bits of accuracy are we talking here? Is this 20? Would you – it's not – you have the same signal noise ratio, one to four – no, four to one. Roughly what – how many bits? It's not a hard – it's not a complicated question, so people are probably way over-computing or something like that.

Student:[Inaudible].

Instructor (Stephen Boyd):It's two, roughly. What? You said two and a half. You believe two?

Student:[Inaudible].

Instructor (Stephen Boyd):You think it's 12? It's two, right? Basically means if I tell you a component – if I tell you the subgradient is this, you can be off by as much as 25 percent. I mean, this is just all hand waving. It roughly means it's about two bits of accuracy. It's not a whole lot of accuracy, right? So that's the point. These are really quite crude. You can see what happens is actually interesting. There's one realization and here's another. Actually, in this one we're really lucky. The errors in the subgradient were such that they didn't mess us up very much, and that's another one where it did, although it can't be stopped. These are big numbers here. That's about – that tends to –

the interesting thing is to get the 10 percent accuracy you probably multiplied your – the number of steps by four or something like that. The really cool part is that you would get – what would happen if the signal to noise ratio were inverted so what if the signal were four times as big as the – suppose when you got subgradient we took this to be, I guess, whatever the .5 – suppose the signal noise ratio were reversed and the signal to noise ratio was .25, not four, roughly? What do you think would happen here? Well, first of all we know the theory says it's going to work. But think about how ridiculous that is, basically. It – you calculate the worst g, that says go in that direction. And to that vector you add a galcion, which is four times bigger. Which means, basically, it's all – it would be very difficult to distinguish between your get subgrad method and this completely random, like, "Which way should I go?" And you're like, "Oh, that way." You know? And it's like, "Really? Can you verify that?" And you go, "That way." It's just totally random. All you'd have to do that like 1,000 times and average them to even see lightly that there's some signal there. Everybody see what I'm saying? What would happen, of course, is that would now mess it up much more.

These would be that. That's what would happen. So this shows you what happens is 100 – you do 100 realizations, you generate 100 stochastic processes, which is the stochastic subgradient method running forward in time. And this shows you the average here. And this shows you the standard deviation. That's a long scale, so that's why these things look weirdly asymmetric. So on a linear scale this is plus/minus one standard deviation. This is also plus/minus one standard deviation, but it's on a long scale. But that's what it is. By the way, it's actually kind of interesting, these points down here correspond to cases where the noises, the noise was kinda bad. Sorry, the noise accidentally pointed you in the right direction, and as a result, you did actually quite well. And of course, these are cases where the noise kinda was hurting you as much as possible. Makes sense? So I guess the summary of this is that the subgradient method you can make fun of it, it's very slow, and all that kinda stuff, but the wild part is actually any zero mean noise added to it does hurt it. And we're not talking noise in the fifth digit. We're talking, if you like, noise in the minus fifth digit, if you want. So you can actually, I mean, which is quite ridiculous if you think about it. Don't try a Newton method when the – when you're calculating your gradients or your Hessians with 25 percent noise.

For that matter, don't try it if your signal to noise ratio is one to four, so it's off the other way around. Okay. So here's a – these are empirical distributions of your suboptimality at 250, 1,000, and 5,000 steps here. And they look like this, and you can actually see these would be the ones at the top of that plot, those arrow bars. And then these would be the ones at the bottom. But you can sort of see that the distribution is very slowly going down like that. So that's the picture. Let me ask one question about this problem. How would you deal with this in a 364a context? Suppose I told you, you need to minimize a piece-wise linear function, but unfortunately, the only method – the source code I won't let you look at. The only thing that calculates this thing only does it to two bits of accuracy. Or another way to say it is every time you call it, you're going to get a subgradient plus a noise, which is as big as a quarter the size of the actual subgradient. How would you deal with that in 364a? I mean, we didn't really have a method to deal with this, but now just tell me what would you do?

Student: Call that function 10,000 times.

Instructor (Stephen Boyd):Right. Right. So 10,000. And, good. Ten thousand was a good choice of number, by the way. So you call the number 10,000 times, and then you'd average those subgradients and what would be the, roughly, how much error is in the one that's 10,000 times? I was hoping for you to say I was going to go down by square root of 10,000. That's why I was complimenting your choice of 10,000, because it had a nice square root of 100. So instead of being an error being 25 percent, it would be .25 percent. So that might be enough to actually run a gradient method. It probably would work okay. So what would happen is at the end game it would kinda start being erratic or something, but you'd get a pretty good answer pretty quickly. By the way, if you evaluate it 10,000 times, I should point something out, this is beating you. So it's not clear – anyway, you're right, that's what you'd do. Okay. So this is actually maybe a good time to talk about stochastic programming. Actually, at some point I want to make a whole lecture on this, because it's quite cool. Everybody should know about it. And it's this. In stochastic programming, you're going to explicitly take into account some uncertainty in the objective in the constraints. So that's stochastic – there's something called robust programming, where you have uncertainty, but you problem it in a different way and you look for worst case type things. But for stochastic that's a very common, very old method, something like that. I should mention, it's kinda obvious that this comes up in practice all the time. So anytime anybody's solving an optimization problem, you just point to any data as – oh, by the way, I should mention this. If you were not at the problem session yesterday, you should find someone who was and ask them what I said. I don't remember what I said, but some of it is probably useful.

So you take any problem, like a linear program, and what you do is you then ask the person solving the linear program, you point to a coefficient, not a zero, because zeros are often really zero. Also one's, those are also not good choices, because a one is often really one. But you point to any other coefficient that's not zero or one and you ask them what is that coefficient. And that coefficient has a provenience. It traces back to various things. If it's a robot it traces back to a length and a motor constant and a moment of inertia, or whatever. If it's a finance problem, it traces back to a mean return, a correlation between two asset returns or – who knows what? If it's signal processing, it goes back to a noise or a signal statistics parameter, for example. Everybody see what I'm saying? And then you look at them and you say, "Do you really know that thing to a significant figures?" And now if they're honest they'll say, "Of course not." And the truth is they really only know it to, it depend on the application, but it could be three significant figures. In a really lucky case it could be four. It could be two, could be one. And, actually, if you get some economist after a couple glasses of wine, they'll look up and say, "We get the sine right, we're happy." So anyway, but until then, they won't admit that.

So okay. So the point of all this is it's kinda obvious that if you're solving a problem, the data, if you point to a data value and – it has a provenience and it traces back to things that probably you don't know better than like a percent. I mean, it depends on the application, but let's just say a percent. By the way, if you don't know any of the data, if

you barely know the sine of the data, my recommendation with respect to optimization – well, my comment is real simple. It's why bother? So if it's really true that you don't know anything about the model, then you might as well just do it by intuition and do your investments or your whatever you want. Just do it by intuition and guess. Because if you don't know anything, using smart methods is not going to really help. So typically you'll know one significant figure, maybe two, maybe three or something like that. And then, by the way, all the stuff from this whole year now starts paying off a lot. And there are weird sick cases where you know things through high accuracy. I mean, GPS is one, for example, where you point to some number and they go, "You really know that to 14 decimal places?" And they're like, "Yes." I mean, I just find it weird. But anyway, normal stuff is accurate between one – zero is like why bother for this. One, two, three, five, six, I guess in some signal processing things, you can talk about 15 bits or something like that, 20. But rarely more than that. Okay. So there's a lot of ways of dealing with uncertainty. The main one is to do a posterior analysis. That's very common. Let me tell you – people know what a posterior analysis is? So posterior analysis goes like this: you're making – it doesn't really matter – let's make a decode – you're making a control system for a robot, I don't care, something like that.

So you sit around and you work out a control system and when you work out the control system, you can trace your data back and there is a stiffness in there and there's a length of the link to and there's an angle and there's all this stuff and there's a motor constant. There's all sorts of junk in there. And they have nut values. And you have a robot controller and you get some controller and now you, before you implement it on the robot – that'd be the simplest way – the first thing you do is something called a posterior analysis. Posterior analysis goes like this: you take the controller or the optimization variable, whatever it is, design on the basis of one particular value, like a nominal value of all those parameters. You take that and you resimulate it with those values, multiple instances of those values, generated according to plausible distributions. Everybody see what I'm saying? And by the way, if you don't do this, then it's called stupid, actually. This is just absolutely standard engineering practice. Unfortunately, it's done in I'd say about 15 percent of cases. Don't ask me why. So in other words, you design a robot controller, you optimize a portfolio, anything you do, you do machine learning – actually, in statistics this is absolutely ingrained in people from when they're small children in statistics. You do this. It's the validation set or something like that. So here's what you do. You design that controller on the basis of a length and a motor constant, which is this – motor constant depends on temperature and all sorts of other crap. You ask somebody who knows about motors and you say, "How well do you know that motor constant?" And they'd say, "I don't know, plus/minus 5 percent, something like that."

You go to someone in finance and you say, "You really believe that these two assets are 57.366 percent correlated?" And they'd go, "No, but it's between 20 and 30 percent correlation, maybe." And you go, "Thank you." Then what you do is you take that portfolio allocation and you simulate the risk in return with lots of new data, which are randomly and plausibly chosen. Everybody see what I'm saying? You change the motor constant plus 5 percent, minus 5 percent. Moment of inertia, change it. The load you're picking up, you don't know that within more than a few percent. You vary those. And

then you simply simulate. And you see what happens. If you get nice height curves, so in other words, that means that your design is relatively insensitive, everything's cool, and now you download it to the actual real time controller. Or you drop it over to the real trading engine, or whatever you want to do. So that's how that works. So that's a standard method. That's posterior analysis. Stochastic optimization is actually going to be dealing with the uncertainty directly and explicitly, and I guess we'll continue this next time. Let me repeat for those who came in late, plea, grovel, and I'm not sure what the word is. At 12:50 to 2:05 today here we're having the world's first, I believe – I haven't been notified from SCPG that's it's not true, the world's first tape-behind. We'll have a dramatic reenactment of lecture one. So come if you can.

[End of Audio]

Duration: 79 minutes