ConvexOptimizationII-Lecture05

Instructor (Stephen Boyd):All right. We'll get started. I guess we'll announce, probably by Thursday, when the actual sorta preliminary project proposal is due. You should be thinking about projects, and you probably shouldn't have completed it by now. But you should certainly have thought about it a fair amount and tossed around some ideas. And you should probably be in communication with us if you're not, just to get a little bit of feedback on what you're doing. And we'll probably make this preliminary project proposal due maybe early next week or something like that. And before we make it due, we will put something on the web, or we'll announce something that's going to be very specific as to the format, because we – it should be two pages and I think we might just even iterate on that this week. We'll just reject it until it's clear enough for us. Clear and simple. So that's what we're going to do. Okay. So let me continue on the material on stochastic programming. So this is going to be one of the examples for stochastic subgradient. So stochastic programming is a – by the way, it's a huge field. You can teach an entire course on it. There are multiple books on this one topic. And a stochastic programming problem looks like this. You have an objective and constraints that depend on some random variable. And in fact, this can – any optimization problem, any one that actually is for some application, is going to look like this to some extent, no matter what. Period. In other words, that any number in the data in your problem will trace back to something, which is not going to be known that well. So in stochastic programming, you assume a probability distribution on these unknown parameters, and you minimize an expected value. And constraints are on expected values. Now expected value might not be what you care about. You might care about things like the variance and things like that.

Actually that's gonna be also handled in the same way. You'll see how in just a minute. So the exact same formulation. So don't think it just means if one of these is – if that's a cost, that's expected cost. You actually care about the variance of the cost. We'll see immediately that this can be incorporated into a framework like this. Okay. Now an expectation, of course, is something like an integral. And obviously, an integral of convex functions is convex. So this preserves convexity. So these are convex – that's a convex function. In fact, I should say something a little bit more. When you integrate like this, generally speaking, these expected values – it depends on the distribution on omega, of course, but if that distribution is sorta continuous, or for that matter, has a distribution that's analytic or something like this, these things actually inherit. They become much smoother. So even if f0 or fi is piecewise linear, for example, or non-differentiable, these are often quite smooth. So that's not relevant. Doesn't mean you can use Newton's method, because, well, you can't calculate any of these numbers, except in very special cases. Okay. Now one sorta very obvious thing to do is to ignore the uncertainty. And you ignore the uncertainty by simply taking for this parameter value its expected value. And that gives you this problem here. That's the – some people call this the certainty equivalent problem. And I've heard, although I don't quite understand it, other people refer to this as the mean field problem. Don't ask me where that comes from or something like that. It has something to do with statistical mechanics or machine learning and stuff like that. But anyway, so this is the certainty equivalent problem. And what it

means is you simply replace this parameter with this expected value and you solve this problem. Now by Jensen's inequality, this is actually favorable compared to that. So when you solve this problem, you'll actually get a lower bound on the optimal value of this one. So that's actually – in fact, that's the way I remember Jensen's inequality. Jensen's inequality says for a convex problem, variation messes you up. Which makes perfect sense. If something curves upward, and you vary around something, when it goes down and it can go up, but the positive curvature says that, on average, you get messed up, which is to say the average cost goes up.

And by the way, when you're solving – if you do anything involving stochastic programming, you should solve this immediately, and then this gives you a lower bound. So now it's not, of course, the solution to this when you calculate the x for this. Although there's a quick way to see how much you've been messed up. I should mention something. This is what's normally done, although not explicitly. So basically when you solve an optimization problem, when you know some parameter's varying and you just ignore the variation, you're just doing this. What you can always do at that point is take the x you find here – this problem, the optimal value gives you the optimal value, gives you a lower bound on the optimal value here. You then take that x and estimate these numbers by Monte Carlo. And if you find that with the x found from here, when you estimate these numbers by Monte Carlo, this number is not much larger than that. And these numbers are not much larger than that. Then you can actually safely presume that you have solved this problem within some reasonable accuracy. By the way, people do that typically informally that's called a posterior analysis. That's a standard method in engineering for dealing with uncertainty is step one, ignore it. Step two, pretend the uncertainty isn't there, then step two is take the design obtained – assuming that all the numbers, the masses, moments of inertia, Young's moduli, all these things, they're all accurate out to full doubles, 12 significant figures – you take that design, and then you subject it to a posterior analysis, which is when you estimate these. Now if this is ten times as big as that, then you back off and you go, "I had a very nice design, but it's not robust." And it means you're going to have to deal with this somehow head-on. Okay. Now there are lots of variations on this. You can actually put in between the expectation and the function any increasing convex function you like, non-decreasing convex function. That's obvious because if you take a non-decreasing convex function of a convex function, that's convex. And this allows you, actually, to shape all sorts of things that you might like. So for example, in a constraint, a very common one is to take the expected value of the plus part, non-negative part of this, and the non-negative part of a constraint is a violation. But it's the amount of violation. So this is the expected violation.

By the way, if fi of x omega is less than or equal to some – if that constraint represents something like a resource usage, it means this is the expected over-utilization. Something like that. It can be all sorts of things. If it is a timing constraint, saying something has to be done by a certain amount, this is called the expected tardiness. If your basic inequality said this job has to be finished by this time, that's the expected tardiness. That works because the plus function is non-decreasing and convex and so on. Now another one would be this, you could take the expected value of the maximum of all the constraints, and this is the expected worst violation. That'll work, as well. So these are common

measures here. And I should mention a few others here. But let me – I'm going to mention a few others in just a minute here, which actually should be on this slide. I'll add them. But the one that's most obvious is this, is a chance constraint. These are called chance constraints. And it looks like this. It's the probability that you satisfy the ith constraint should be bigger than eta. And you should think of eta as, you know, 95 percent, two nines, three nines, this kind of thing. I mean, that's how you'd imagine this. So basically it would say something like this: optimize this thing for me, but what I want to do is I want the probability that my link goes down or has inadequate signal to noise ratio should be – I want the probability that I have inadequate signal to noise ratio to be something like 99 percent or something. At least 99 percent. And that would be something like, by the way, in lots of contexts this would be called – more specific contexts – this would be called something like an availability constraint. If, for example, this represented a communication link having adequate signal to noise ratio or something like that, this would be an availability constraint. You're saying that link has to be available at least 99 percent of the time. Or it's an average constraint. Just flip it around. Here's the bad news. Bad news is that these are very nice constraints. You can write whole books on them. But generally speaking, they're intractable. And they're not convex. However, you know one case where this is. I don't know if anybody remembers it?

Student:[Inaudible].

Instructor (Stephen Boyd):What's that?

Student:[Inaudible].

Instructor (Stephen Boyd):Precisely. So when f is affine, it's actually biaffine, and it is affine in x and in omega. In other words, it's a linear constraint with uncertain coefficients. That's the most basic one. Then this constraint right here is a second-order cone constraint. And that's actually when you think of a second-order cone constraint, you should almost think if – I mean, when you think of second-order cone programming, you can almost think of it as chance constrained linear programming. That's what it is. So that's one case, but in almost all cases, these are not convex. Now turns out there's all sorts of good uristics for doing this, anyway. Let's talk about a few more things up here. There's actually one more I should add here. How about adding an almost sure constraint? How would you do that? Let me write down what that is. An almost sure constraint – let's write down this one. Suppose what I want is fi. Can I handle that and is it convex? That's the question. Of course, obviously, most cases it's going to be infeasible. So it's gonna have, you know, if your distributions have infinite or galceon have – so infinitely bad things can happen with small probability, constraints like this will just make the problem infeasible instantly.

But in some cases, this would make sense. And the question is, can you handle it? Can you use this mechanism, and if so, how? How do you do that? Can you write that as the expected value of something? What do you think? What would do this? You could write it this way. It's just the expected value of the indicator function. So this is, in fact, less

than or equal to zero. And the indicator function here, this function is the following function: it's zero, if you're negative. And if you're positive, it's plus infinity. That's increasing in convex. Okay? Everyone agree? I mean, it's rather extreme. It goes from zero to plus infinity, but the point is that's increasing in convex. This function is, in fact, if you apply this to f, you can pose this function with f, all that happens is you redefine f to be plus infinity if you violate the constraint. If you say the expected value of that is less than zero, it's pretty much identical to saying that this is true almost surely. Okay? So you can handle almost sure constraints. And I should mention one more that's kinda widely used. Let me see if I can get it right. I think I can get it right. I might – I shouldn't do this because I'm going off script, or whatever, but that's okay. Here it is. A very common version of this is the following: you take x of something like that, so that's clearly increasing – gamma's a parameter, a shape parameter – x, of course, goes up very steeply, and so this gives you an increasing function. And then you simply put an expected value here. Okay? Now, in fact, you can also do one more thing. That's convex, but then so is that. And I'll just leave it this way. And we'll see why in a minute. So if you do this, and let's just look at the objective for this. So if you look at this function here, first of all, let me go over a couple of things.

I'll just say this briefly because, in fact, it's likely I'm going to make a terrible mistake and regret that I've gone off on this tangent. But anyway, let me try it. Everyone knows that log sum x is convex and non-decreasing. This is like a generalization of log sum x, because it's log of expected value of x. So this is, in fact, a convex function here of x. Now the question is – oh, I'm sorry, you put a gamma in front. That's just for the normalization. Minimize – actually, you don't need the log, you don't need the gamma, because that's convex right there. This just makes it prettier. Now it turns out if you work out what this is, if you do an expansion on what this is, the first term is actually gonna be something like the expected value of f0. That's as if you just minimized expected value of f0. The next term in the expansion of this – I guess this is the moment – not the moment – which one?

Student:[Inaudible].

Instructor (Stephen Boyd):Yeah, it's the – there's the moments and then there's the central moments, or whatever, the centralized ones. This is whatever –

Student: Cumulate?

Instructor (Stephen Boyd):It is gonna be something – let's call it the moment generating function, except it's not the moment generating function. Maybe it is the cumulate generating function. Good. The first term in this is expected value of f0. The second term is the variance, and that scales with gamma. So when you see this, this is just a general measure for risk aversion. It's a general construction for handling risk aversion. So let me explain that. Make a note. We're gonna add this. By the way, although what I'm gonna say now is gonna be wrong, possibly wrong and all that, I don't guarantee any of it, by time we stick it onto the slides here, it will be correct or closer, anyway. So let me give an example of this. If f0 is what you want to minimize, for example, it's negative

revenue in some finance application, so it's a negative revenue, then you'd say, well, if you just do nothing here, if you just do the basic thing like this, you're asking please maximize my revenue. This is just – if you just take the expected value it says maximize my expected revenue. But then the variance is irrelevant, so you might actually find some point which maximizes the expected revenue, but as huge variance. And you might care about that. And you might say, well, no but I want to trade off the expected revenue and the variance, as well. You would do something like that and then gamma is called the risk aversion parameter. And as you tune gamma, you would trade off expected value versus variance in the first order. So okay, that's just – like I said, you can make a whole – we could easily make an entire lecture on it. You could make a course on it. And there's multiple books on it. Okay. So let's look at how this works with stochastic subgradient. Suppose you have a function that's convex in x for each omega. And I have a subgradient that's g of x in omega. That's a selection of a particular subgradient, subgradient with respect to x of the function f. Little f is going to be the expected value of this thing.

You average out over the omegas. And that's gonna be convex. Then a subgradient of f is nothing but this. A subgradient is the expected value of that gradient selection function over there. And now comes the interesting part. Suppose you generate M samples from the distribution and you evaluate capital F of x omega i. Now if you average capital F of x omega i, if you just average that, that's an unbiased estimate of little f of x. And in fact, you're gonna do this anyway, because you're gonna want to estimate this number. So this is something you're gonna have to do whether you like it or not, because when you say here's my number, you wanna test what's happening, you wanna test your yield or whatever your value is, you're gonna actually do this anyway with capital F here. But it says that if at each point you've calculated subgradient and then averaged the subgradient, then it turns out this thing is actually a noisy unbiased subgradient. Okay. So by the way, this fits my meta-assertion that if you can evaluate a function, you can evaluate a subgradient. And by the way, this is an example where technically you cannot evaluate the function, because basically, you can't evaluate this exactly except in a handful of silly cases.

I mean, they come up and they're used, right? You know all the things you can – I mean, basically how many integrals can you calculate exactly? The answer is 15 or 17 or whatever the number is. But the point is, in general, you can't do it and you're gonna have just by Monte Carlo. So here's a case where what you can do is by Monte Carlo you can compute this number approximately, and as accurately as you like, and at the same time, with very little effort, other than calling f dot subgrad here or something to get subgrad here, you can actually also calculate an unbiased subgradient. Okay. So let's do an example and see how this works. We're gonna minimize the expected value of a piecewise linear function. Now what's interesting here is that the coefficients ai and bi are going to be random. So a piecewise linear function. I want to minimize it, but the expected value – but ai and bi are random and you're gonna commit to an x. If you wanna visualize what goes on here. You commit to an x and then you'll pull random samples of the ai and bi. You'll evaluate this function and then you'll take the expected value of that. And that's gonna be your value. I mean, if the ai and bi are over the map, it's not that interesting, this thing. If the variation in the ai and bi is in the eighth

significant figures, it's also probably not that interesting. It's interesting somewhere where they're in the 10 percent range, 5 percent range, 20 percent range, something like that. That's where the value of being smart – there's actually a positive value to being smart. If the variation is just all over the map, there's nothing you can do except, I don't know, take x equals zero or something like that. If the variation's tiny, you just ignore the uncertainty altogether. Somewhere in the middle you can actually do a good job.

Okay. So what we'll do is I'll explain this. We'll do a couple things. We're going to compare this certainty equivalent. In certainty equivalent, what you do is you simply minimize one piecewise linear function. So that you do by linear programming. You just plug in the expected value of a, expected value of b. This will give you a lower bound on the optimal value here. And there's a lot of uristics you could do. But here's one. One is this. Let's see – we'll plug in the expected value of b and what's interesting here is when you take the expected value, let's ask about variation in a. Variation in a gets multiplied by x here. By the way, this one – oh no, this does matter, because the i's have different bounds. Variation in a gets multiplied by x. So here you might, basically you want smaller x is gonna be less sensitive to variation in a. So you just add this. It's a uristic. There's lot of other uristics, by the way. Okay. So here's a problem instance. And let's see how these things look. So the ai, those are the row vectors, and then I guess they have that there – how wide are they? They're 20 wide. I mean, each ai is – there are 20 variable.

I guess each entry has a standard deviation of square root of five. And b has also very substantial – sorry, let me see if I've got this right. That's right. Then ai bar – so these are – actually, I think these are ai bar here, I'm guessing, although it's probably about the same either way. Okay. So here is the certainty equivalent just ignores everything and you get, let's see, you get – where am I? Here we go. This is the certainty equivalent problem. And then this shows you the distribution of values when you just do Monte Carlo here. So this is Monte Carlo here. And you can see that there's all sorts of values here – these are very unfortunate things when you committed to an x and then the a and the b were wrong for it. This is the uristic. And you can see that this is the distribution here. And you can see that the uristic is already doing a lot better. I guess this is the mean of this. So it's an estimate, well, modulo 100 Monte Carlo samples. That's an estimate of the expected value of the objective value here. The uristic is done well, because it's penalized large x and so you've moved away from that or whatever, and you get a tighter distribution here. And then this is f stochastic and this is computed using the stochastic subgradient, something like that. So this is presumed to be something close to the global optimum. And these dashed lines I am gonna guess the following: they're a mystery. We can guess that that's the optimal value. If that is an optimal value of this, then that is a lower bound on this problem. The stochastic subgradient gives you something there, and so if I'm on the stand and I'm being deposed or there's lawyers present, I could only say something like this. This, again you have to believe in Monte Carlo, but modular Monte Carlo, that's an upper bound on the optimal value and this one here is the lower bound over here. It's a mystery as to what these are. I'll have to find out what they are.

Okay. By the way, if you're doing stochastic optimization, this is how you judge one x or another, is you – you're gonna want to run histograms anyway, and so you're gonna want to look at things like this. Okay. This is 1.34. There we go. All right. So let's see, if you run the stochastic subgradient for a very long time, this is sorta what happens. You get, I guess the theory says it was gonna converge to the optimum always. It's just a question of how long it's gonna take. And you can see here in this case you're taking 100 and you're taking 1,000 Monte Carlo samples to produce the expected subgradient. Actually, in those cases, what's kinda – you can almost argue in these cases that you're actually computing the gradient of the function almost exactly. So these are the gradients. And in fact, this is what sorta the gradient stuff does. It goes like that. And it's gonna keep going down. And I guess it's estimated to be something like 1.34. And this is how close you get to the optimal value. So that's that. By the way, this is not a – depends on here the effort in each of these is running the Monte Carlos. So in fact, these things, the green one here, for example, is 1,000 times more costly than the other one. So if you actually write these in terms of – if you actually count these things, then you'd find probably that either this one or this one is gonna be by far the best one here. In terms of if you rescale the axis not to be in iterations, but actually in total number of Monte Carlo simulations, then actually this one, somewhere in here is gonna end up being the best one. Almost certainly this one here, something like that. Okay. So that's Monte Carlo estimation. Is that on your current homework? It's on your next homework. Will be. Homework 4. Good. Okay. So we'll look at another area where this stuff has been discovered and so on. It's called online learning and adaptive signal processing. So that's something. Some of you have probably seen this in some contexts. So let's see what this is. This is the simplest possible problem. It goes like this. We have x and y where x is a vector and y is a scaler, and they come from some joint distribution. And the simplest possible problem is something like this, is to find a weight vector in our end for which w transpose x is a good estimate of y. So that's what you want.

These come from a joint distribution, so you don't know – well, we'll talk about that in a minute. So one way to do this is you choose w, minimize the expected value of some convex loss function like this. Now, by the way, if the loss is a square, then this is a mean square error, and all you need to know are the second-order statistics of x and y to trivially write out what the estimator is. And it would be linear and it would be – sorry, actually in this case, over all estimators, in fact, the optimal would be linear or affine or something like that. Okay. But for example, if you take the loss to be absolute value, there's going to be no simple analytic description for this. It's gonna depend on the distribution. It's certainly gonna depend on more than the second moments of the distribution. Okay. And the simplest context for this comes up in adaptor signal processing. What happens is you're given a sample at each time step. That's the right way to do it. So you get a vector x, you get a vector y. That's all. And you want to come up with a vector w. That's all. That's what you want to do. That minimizes some loss function here. Okay. Now let's see how this works. We'll assume l is differentiable or whatever, if you take a subgradient otherwise. It doesn't matter.

If you have a noisy unbiased subgradient, a single sample gives you that, because what you do is you take l prime or actually any g(k) in the subdifferential of this thing here,

and you get l prime multiplied by x(k + 1). So that is your – that's a noisy subgradient and it is a – so the stochastic subgradient in algorithm just looks like this. You step in this step. Now here you make a step in the negative subgradient direction. Now if the loss is a square function, this gives you the lms algorithm, which is that you update in the direction proportional to your error. By the way, this is your error at the kth step. That's your prediction error, basically. And you look at your prediction error and based on your prediction error, you will update your waves. So you don't take a full step here, because in fact, these are all stochastic and even when w is optimal, you will be making mistakes here. So that's why you need this to be damped here. Okay. When this is absolute value, you get something that's called – I guess it goes back into the '70s, maybe, is this so-called sine algorithm. And the sine algorithm is very simple. It says calculate the prediction error, calculate the sine of the prediction error, and then bump up or down the component of the weight dependent on the sine. So here this thing is just a plus/minus one vector. Okay. So here's a quick example. You have a problem instance with dimension 10x and y is a scaler and these have zero mean and they have some covariant, and the expected value of y is – well, the variance of y is 12, so the standard deviation is square root of 12, or three and a bit. So let's see how this works. And these are just running this algorithm here. And this is with step length one over k. And you can see what's happening here. At first you're making, at least for 20 – for the first ten steps you're making horrible mistakes, horrible prediction errors. But after about 50 or maybe 100 steps, you can see that your error has gone down. As to whether or not this has reached statistical equilibrium, I don't know. It might have. Maybe not quite. But it looks like it's approaching statistical equilibrium. When this thing converges completely, of course it's still gonna make errors, because even with w*, the optimal w, you will make errors in prediction at each step. And that, of course, depends on the properties of the joint distribution and so on. Okay. So here's the empirical distribution. This is over 1,000 samples. This shows you the distribution error for w*. This is what we presume is the optimal w. And this is what it looks like, so it looks like you can predict with something like this.

Remember that y itself varies over – has a standard deviation of about three and a half. So this maybe has a standard deviation of, I don't know, whatever, half or something like that. But the point is, you can actually predict it within about 15 percent, is the way – when you have the optimal weights. You're predicting at about 15 percent. You cannot do any better. That's what it means to be w*. Okay. So I think this finishes up this – actually, you will see some of this on your next homework. Any questions about this? Okay. I should mention we're going over this kinda fast, obviously, in lecture. So there are notes on this. They flesh these things out in a little bit more details. Of course, there's also homework on it. Not a big deal, but just one so you can say you did a stochastic optimization problem if anyone asks you. By the way, my suspicion is a lot of people who've even written books about it haven't actually really done it, so you'll be ahead of them on that. That might be an exaggeration, but not much of one. We're gonna start a new section of the class, which is on methods that are in spirit like subgradient methods. They require only subgradients, and absolutely nothing more than subgradients. But they are more efficient. Certainly in theory. Some of these methods are actually efficient, so we'll see that. We'll get to some methods that are actually just, according to theory,

they're efficient, they're polynomial time and all that. They're quite slow, but they have uses. Some of them actually work quite well. We'll get to some of those.

And they're all based on the idea of localization and cutting-planes and actually, these are beautiful, the ideas are beautiful, they have lots of applications and distributed optimizations and distributed design and it's good to know about these things. Okay. So what is the idea in a localization method? In a localization method what happens is – you know what these are? Let me explain it. These are basically generalizations of bisection in R. So bisection on R, how does that work? Well, it works like this. You have an interval and you know the solution is in this interval. So you maintain an l and a u. This is bisection. Then you do something like you check the middle, and then by analyzing something in the middle, like the derivative or something like that, what you know is now – after you know that, you say that actually the solution is either on the left or the right. This is the essential idea of bisection. And so at the next step you have a new interval, which is exactly half the size of the interval before. So that's bisection. We all know about that. Okay. Now the cool thing about bisection is that your ignorance, as measured by the width of the interval, goes down exactly by a factor of two at each step.

Or, if people who are actually trained in information hearing allow me to say this, you get one bit of ignorance reduction per step. Because at each step the width of your ignorance as to where the solution is goes down exactly by a factor of two. So you gain exactly one bit of information about the location of the optimum. Okay. Now this is bisection. Then you might ask yourself, maybe you have already, is there an analog of bisection in R2 or in Rn? I don't know if you've ever thought about this. Have you thought about this? And it's complicated. It's very complicated. And you could easily convince yourself the answer is no. And the reason would go something like this. That R, unlike R2, R3 and so on, is ordered. It's the only one that's ordered. You might say you can only bisect – that's a very special thing. It only happens in one dimension because it's ordered. Actually, what you're gonna see in the next couple of lectures is that's false. You can actual do bisection in Rn. It's not gonna look exactly like bisection, and it's not gonna be as good. You're not gonna get a full bit at each step, but you're gonna get something like it. You're gonna get a guaranteed minimum fraction of a bit. And it's gonna be more complicated, because instead of an interval localizing your solution, it's gonna be some general set. So that's the super big picture. This is gonna be bisection in Rn, which is actually, if you think about it, pretty cool. And the way it's gonna work is so localizing is that you're gonna maintain a data structure of some kind or some set, and it's gonna summarize what you know. And it's gonna say something like this: my solution is in that set. Exactly like in bisection, where you have an interval and you say, you stop bisection early, you terminate it and you say what you know. You have an l and you have a u. And the optimal x* is between l and u. And u minus one is a numerical measure of your ignorance as to what x* is and so on.

This will be the same thing, except the data structure is not gonna be an integer, that's for sure. These methods only require a subgradient of the objective or constraint function at each step. So these are gonna require only a weak subgradient calculus, so if you implement methods that calculate a subgradient, that's fine. Anything you could use for a

subgradient method you could use for localization methods. And these directly handle non-differentiable, convex. They'll also do quasi-convex problems. Now these are actually gonna require more memory computation for step than subgradient method. Subgradient methods require nothing. They basically call a subgradient and then, in fact, how many of you know about blast? Blast? I know it's early, but a few more now. Okay, a handful. So basically what's the computational effort in the subgradient method? We never really talked about it. It's gonna be a very short conversation, so let's do it. What's the computational effort of the subgradient method? What do we have to do at each step to get a subgradient?

Student: [Inaudible].

Instructor (Stephen Boyd):Okay, then what do you have to do? What computations do you have to do?

Student:[Inaudible] vector product and then add two vectors.

Instructor (Stephen Boyd):Okay. How do you say that in blast?

Student: Axby?

Instructor (Stephen Boyd):Yeah, axby. Daxby, I guess. It depends on what you want to do. So in other words, it's about the most pathetically small amount of – it's an order n update. n, by the way, is the number of variables. So if you merely – that means basically it's the effort of just sorta writing to the variables and so on. So it's nothing. Just accessing the variables, doing one or two flops per variable. That's all. So a good approximation of the computational effort per step of the subgradient method is nothing. All the effort is in calculating the subgradient. And the other one will be free. Okay. So these are gonna require – we're actually gonna have real calculations here. These can be much more efficient. In theory, for some of them, and in practice for some, and they don't completely coincide. So we'll see ones that are efficient in theory but not practice. We'll see – no, that's not true. All the ones that actually work well in practice actually, I'm pretty sure, are also efficient in theory. But the sets are not gonna be the same. Okay. So it's all based on the following idea. You have an oracle, and it's actually better – well, one way to do this is to abstract this away from subgradient. Subgradients will come in later. But you have an oracle, and your oracles gonna go like this: what we're gonna do is we just want to find a point in the convex set, or determine that the set's empty. And the only access to x is gonna be through a cutting-plane oracle. And what the cutting-plane oracle is, it's gonna go like this: you're gonna query or call the cutting-plane oracle at a point x. And what's gonna come back is either a token or something that tells you you're in the target set, in which case, by the way, you're done. Or it's gonna come back and give you a separating hyperplane between little x and big X. This is called, actually, cutting-plane or a cut, since it eliminates all half space of x's from consideration. So let me show you what this looks like. Actually, let me just draw a picture, just to show you how this works or what it looks like. In fact, this one you have to draw pictures for everything. So the way it works is this. There's some set here. We don't know this set.

This is known only – so we don't have access to this set. This is capital X. And then what happens is all you can do is actually query an oracle at a point. So if you query an oracle at this point, the oracle simply returns a token that says you're done, or it's in it. However, if you query the oracle here, the oracle has to do the following, it has to return a cutting-plane, and a cutting-plane is anything that separates that point from this set.

That's a cutting-plane. And the idea is basically when you know that – what you know is that this entire half space cannot contain any point in here. And therefore, there's no reason to look in this half space anymore. When you get this half space the exact semantics is this. And by the way, it does not tell you that this set – this set could be empty, in which case, by the way, you're allowed to return the semantics, the agreement there, just to conform to the semantics. If the set capital X is empty, you can return any cutting-plane you like, because it is true that there is no point in here that is also in your set, because actually there's no point in your set. Okay. So that's how this works. So this is the idea, and by the way, this is something like bisection. Because in bisection, what happens is you query at a point on the line. This is bisection. And in fact, what bisection tells you is it says either the solution is there or there. It actually returns – that's a cutting-plane. And cutting-plans in R are a little bit boring. They're basically sorta a point saying it's either to the left or to the right or something like that of some specific point. So that's a cutting-plane. And if you want to think in your head that this is something like you're getting one bit of information, which is not quite right, you're welcome to, because it kinda gives you the idea. Now this is an idea of a neutral – by the way, if this is the point you query at, you can actually return lots of cutting-planes. Obviously any plane, there's tons of planes that fit in between this point and this set. Any one of them is an absolutely valid cutting-plane to be returned. I mean, you could return this, for example. That would be fine. You could return something like this. It just doesn't matter. By the way, if this cutting-plane goes through the point, the query point, then it's called a neutral cutting-plane. So it's called a neutral cutting-plane, and that looks like this.

So here is your point x where you query, and what comes back is a neutral cutting-plane, so it basically says do not look to the right here. If this set even exists, if it's even non-empty, it's entirely contained on the left here. So a neutral cutting-plane goes through that point like this. A deep cut is one where you actually slice off – actually, not just x, but like a little ball around x is what it means. So it basically means not only is your point not optimal, but in fact, it makes a little ball around it and says it's not optimal, too, because if this is deep, it says that there's a little ball here that is not optimal – sorry, not in capital X. By the way, there's also an idea of a shallow cut. A shallow cut is one where you get something over here that's not really a cut. Thos come up, too, although we won't look at them. Okay. So let's do unconstrained minimization. So let's do this. We want to minimize a convex function. We'll let capital X be the set of optimal points, so any minimizer. And what we're gonna do is this: we take the basic inequality for a subgradient, then that says, of course, that if you evaluate a subgradient at x, you get a cut, you get a cutting-plane. And that's actually sorta the point, is that if you evaluate a subgradient, you immediately get a neutral cutting-plane at that point.

Basically the way that works is this, you have the sub-level sets. Let me draw a function here. So here's some non-differentiable function. We query at this point here, and you get a subgradient that looks like that. There's many subgradients, but they all give you a cutting-plane for the minimizer. Because when you get a subgradient, what you're told is that for this entire half space defined by the subgradient passing through that point, this half space, there's no reason to look anywhere here, because every point here has a function value larger than or equal to the function value there. There's no reason ever to look there, because you'll only find worse points than this one. So that's a neutral cutting-plane. Actually, the fact is you can also get a – this is the picture which I just drew anyway – looks like this. Basically what you should think of is this: when you evaluate a subgradient, you are actually getting a cutting-plane with respect to what the solution is. Now you can actually get a deep cut, and deep cut goes like this. This is much more interesting. It goes like this. Suppose you know a number, f bar, that is bigger than f*, but lower than the current value? And one common choice of f bar here is actually the following: this f bar can be the best point found so far. That's what we called before f best k. And there's no reason – if the last point was your best point, then of course, these are equal and none of this holds. But if the current point is worse than the best point you found so far, it means, actually, it tells you that your current value is bigger than that, and then this says that if f(z) is bigger than f(x) plus this thing, you must have this. If this holds, then you have f(z) is bigger than f bar, which is bigger than f*, and that says that you're not on the optimal point. So you have a deep cut, and the deep cut is this. So and there's lots of other examples of this. If you knew f*, for example, that would be another example. You just put f* right here and you'd get a deep cut immediately. So these are – that's a deep cut. Okay. In a feasibility problem, you'll take x to be the set of feasible points, and if x is not feasible, so here's how you generate a cutting-plane. Someone gives you x, you cycle through all the fi's. If all of them are less than or equal to zero, you have to return a short token saying done, x isn't in capital X. If not, it means that one of these fails. So in fact, you could even stop at the first one will generate a cutting-plane.

So the first time you get a violation, what you do is you evaluate a subgradient and you get a – here you can actually do a deep cut, and this gives you a deep cut. So that gives you a deep cut. Here's the deep cut is right there. So that's how this works. So the idea is if you want to solve a feasibility problem you cycle through the constraint functions. If you get to the end of the list, everybody's satisfied, done. You're feasible, it's all over. Otherwise, you take any violated constraint. The first one you encounter, for example, call get subgradient on it and return a deep cut plane at that point. Okay? And if you have an inequality constraint problem – so actually what I'm going now is I'm explaining how to create a cutting-plane oracle, how to wrap an oracle method around various problems, like feasibility, unconstrained optimization, constrained optimization and so on. So that's what we're doing here. How do you do an inequality constraint problem? That's easy. If x is not feasible, then you get a feasibility deep cut. And what you do is if someone gives you an x, you cycle through all the constraints. If you reach the end of the list, you set a flag saying that x is feasible, and you evaluate f0.

By the way, when you evaluate f0, you possibly will update the best value found so far. Best feasible value found so far. However, if you don't make it through your list of

constraints, you stop at the first violated constraint here, you get a subgradient for the violated constraint, and you return this deep cut here. And this deep cut is very simple. You could actually not only return the deep cut, but return an annotation or a comment to the caller, to the oracle caller, although it's not required. The semantics doesn't require you to give a comment back. You can give a comment, which is basically like here's a deep cut, and if someone asks you why is that a deep cut, you can say, "Well, because anybody who doesn't satisfy this inequality here, I guarantee is gonna violate the 13th constraint." That's what it means. You're not required to do this, but you could do this. Okay. Now if all of these are correct, if x is feasible you just have a neutral objective cut, or a deep cut if you have an objective value like that. So now you know how to generate a cutting-plane oracle for pretty much any problem you would encounter, constraints, inequality constraints, all those sorts of things. And now I can give you a completely generic localization method. It goes like this – it's conceptual because that means something in here is vague. I'll get to that. It works like this: you start with an initial polyhedron known to contain x. It doesn't have to be a polyhedron. It can be any convex set known to contain – it can be a ball or something like that – known to contain initial point. Sorry, to contain this, which is no guarantee that this is not answered. Okay. And it works like this: you choose a point inside this polyhedron. Now this part is vague and we're gonna get to that in great detail.

So you choose a point in this polyhedron and you call the oracle at that point. If the oracle returns to token, the success token, done. Algorithm's over. You're done. If it doesn't, it has to add a new – it has to return a cutting-plane, and you simply update your polyhedron. So the idea of the polyhedron is it summarizes what you know at each point. Or another way to say it if you want to think of it from the negative point of view is it summarizes your ignorance, is what the polyhedron is. Because basically, the true semantics is this: Pk at step k or whatever, if something like it's the set – that's all you can really say about where capital X is. All I can say is capital X is a subset of Pk, nothing else. Now if that's empty you can certainly quit and then you can certainly say the problem is infeasible. And you said k equals k plus one. So here's the picture. And the picture is draw here for a neutral cut. So here's the way it works. Pk – and by the way this is nothing but a generalization of bisection. In bisection you have a polyhedron in R. Polyhedra in R – we can have a very short discussion of the theory of polyhedra in R – polyhedra in R are intervals. So it's not that interesting. Okay. So here you have a polyhedron, you query at this point here, and a, in this case, a neutral cutting-plane comes back and it says that it points in this direction. And that says you have just ruled out all of this stuff here. This is impossible. And so that's your old set of where capital X might be, if it exists. That's our new set. Oh, and in fact, you could talk about all sorts of measures, but a very good measure would be – and absolutely defensible, I think, in information theory – would be something like this: the log of the volume of Pk is a very good measure. If it's log based two, it's a very good measure of sorta the actual number of bits of information you've gained here. And so, actually, I'm gonna ask you for this particular iteration that went from this to this, how many bits of information did we obtain?

Our ignorance went down by a certain number of bits. What's the number? Twelve? 0.1? What is it? Just eyeball it.

Student:[Inaudible].

Instructor (Stephen Boyd):Yeah. If it was one bit, it means that volume went down by a factor of two. Or the area, it's the area. Does the area look like it went down by a factor of two?

Student:[Inaudible].

Instructor (Stephen Boyd):More than two. Did it go down by a factor of four? No, probably not. So one and a half was a perfectly good estimate. So our ignorance just went down 1.5 bits, roughly, here. And actually, let me ask something for this problem. What's really interesting about this now is when we queried at this point, we had no control over what cutting-plane was gonna come back. And so let's actually draw some pictures about what could have – what's the best thing that could have happened, what's the – well, let's think about that. I don't have to have the same set. Oh, here it is. Okay. So there is it. And I query at this point. And let's do – first, let's do deep cuts. All right. So I queried the oracle here. And what would be the best thing that could possibly happen with a deep cut? My lucky day. I have tipped or bribed the oracle and what happens?

Student:[Inaudible].

Instructor (Stephen Boyd):Yeah. So the oracle comes back saying if there are any feasible points or any points in capital X, they've got to be – I guess it points that way. That's our protocol. It's that way. So basically it eliminates all of this and leaves only, quite conveniently, this one vertex. This could happen. At which point, by the way, because really literally, just that single point there, we can now say with certainty, we're done. Well, I'm sorry, I have to call the oracle at that one point, and then have it return to me the token saying you're in. Right? So that's the – so this is the best thing that could happen. And in fact, let's suppose it wasn't quite that good, but it let a little tiny baby triangle here. But in this case, what's our ignorance reduction ratio? Huge. Absolutely huge, right. I mean, it went down by – I guess if it went from the whole thing to a point, then our ignorance reduction ratio was infinite, because we went from something with some area to zero area. So arbitrarily good things can happen. Okay. Well, it's silly to base an algorithm on that. Not only silly, but it won't work. So let's talk about what's the worst thing that can possibly happen? Well, the worst thing that could happen is you're gonna get a neutral cut. And let's talk about the direction of the neutral cut. If the direction of the neutral cut in this case looked like that, how many bits did you get? Just eyeball it.

Student: Two.

Instructor (Stephen Boyd):Two, three, something like that. You got a couple of bits. You were lucky because you're getting a full two bits. By the way, this is sorta like – you're now beating bisection. In bisection, if you query at the middle and you go down by a factor of two, you get exactly one bit per iteration, in ignorance reduction you get one bit. Here you were lucky. But suppose, in fact, it came back like that. So your prior

knowledge – although really probably the correct name for this thing is your prior ignorance – your prior ignorance is this polyhedron here. And your posterior is this thing with this part cut off. And so what's your ignorance reduction now? Way under a bit, right. In fact, if you think that this was, I don't know, one-eighth the size of that or, I don't know, one-sixth or something, so it's between two and three bits, then it basically says the area went down only by a factor of seven-eighths, so you did quite poorly.

Student:[Inaudible] if you cut doesn't go through the polyhedron?

Instructor (Stephen Boyd):No, no. You have to – you're always gonna gain something. That's your guarantee. Well, no, no, hang on. Let's look at some query points. Let's now discuss query points. So let me ask you this, suppose I query here. What can't happen if you query there? It can't come back and say, oh, you got it. Otherwise, something is wrong. Because basically, according to all the contract, I'm supposed to be able to certify that any point that's actually in capital X is in this polyhedron, and I called the oracle there and it said, yep, that's it. So something is wrong. However, could something good happen calling here? Could I call the oracle there and have something good happen? Yeah, absolutely. What could come back could be this plane here, in which case I say, great, I win. I got it in one step. However, bad things could happen, and I'll show you one of them. You could simply call at this point and get this thing back, and how much ignorance have you reduced now? None. So okay, so basically querying outside is, at least in the worst case, as dumb as you can get, because if you query outside the localization region, there's a chance that your ignorance won't go down at all. So how about querying here? And let's do neutral cuts. If you query there – by the way, if you're lucky, what's gonna happen? Neutral cut through that point, what happens? It's gonna go way down. What's gonna happen is it's your lucky day, you point in that direction, you have a huge reduction in ignorance. Okay. On the other hand, it could just as well have worked this way. And if it worked that way, your reduction in ignorance, though positive, is extremely small. So now you begin to think, if you wanted to do something that's gonna work well, even if the worst case, you obviously want to query this somewhere near the center. I mean, that's clear.

And in fact, what you'd really like would be this. You'd really like to have a point where no matter how a plane goes through that point, the area is divided. In fact, if you could find a point in that polyhedron where every hyperplane that passes through that point divided the area 50/50, that would be perfect, because then at each step your ignorance goes down by exactly 50 percent. Everybody following this? That would be the exact analog of bisection, because in bisection your polyhedron is an interval, you query at the middle, and the posterior polyhedron is either the left or the right half. Your ignorance goes down by exactly one bit. So that's it. Now the question is, is there a point in a polyhedron for which any plane passing through it divided the polyhedron 50/50 in volume? And the answer is, sadly, no. There's no such point. Although that was a great idea, it's not gonna work. So that's not gonna work. So we're gonna see, though, there's points that are almost as good. That's the wild part. So it's not gonna be as clean as that. If it were that clean, it would be actually quite cool. But sadly, it's not the case. So all of this is the argument for why we would want to pick this near the center, and that's

because we're risk-averse. By the way, if you want to make an algorithm that's risk-seeking, go ahead and evaluate it near the boundary. Go ahead if you're gonna trust it. If you're just punting at that point, go ahead and query it outside and tip or bride the oracle or something. Okay. So here's just a picture of how this would update typically. So here's Pk, here's this point. This shaded part has been ruled out. Let me see. Sorry, pardon. I've got it now. This is showing if that's your point, if you choose to query here, this is what happens. The left column shows maybe good things and the right column shows bad things.

And in this case, your ignorance reduction is about one bit in each case. And in this case, it's considerably more than a bit, and considerably less than a bit here. Okay. So we talked about this already. The most famous example by far of a localization method is bisection on R, Pk is an interval. Obvious choose for the query point is the midpoint, and your ignorance goes down exactly by a factor of two at each step. So that's the bisection algorithm. It totally obvious. But it's a member of this family of cutting-plane methods. So that's it. And, in fact, you can say all sorts of things. So the uncertainty is half, so you get exactly one bit. We've already said that. The number of steps required to get an uncertainty in x less than R is this: it's log two of the length of P0 divided by R, and that's log two of the initial uncertainty divided by the final uncertainty. So that's the number of steps you're gonna require. That's exactly the number of steps required. So that's what it's gonna be. Okay. Now we're gonna talk about specific ones. The main difference in all cutting-plane methods is how do you choose that point xk. We'll get to look at least at one. So here's some methods. One is the center of gravity method. Another is the maximum volume ellipsoid cutting-plane method, Chebyshev center, analytic center. And we'll go over all of these. We'll go over this one in some detail, because this one, actually, has the interesting attribute of actually working extremely well in practice. We'll get to this one. This is a beautiful one. It's the most beautiful one by far, and it has one minor drawback that we'll get to today. And then these actually also – this one, at least, works well, as well. So this is CG algorithm. It goes like this: you want to query in the center of your polyhedron. So we're gonna take the CG of that polyhedron, and that's the integral of x divided by the integral of one over that thing. And then it turns out there's a basic fact that's actually quite extraordinary. It's not hard.

We should put it possibly on the homework, because it's a guided proof that's very short. It goes like this: it's says basically if you take any polyhedron in any dimension and you take the center of gravity of it, and you put a hyperplane going through that point, you divide your polyhedron into two regions, and the question is, how unevenly can that divide the volume? This is Rn. And the answer is you can never be worse that 63/37. Period. Ever. Isn't that insane? Notice it has nothing to do – and by the way, you can sharpen this for dimension – well, for dimension one, actually, this is 0.5, because it's the center. For dimension two there's another number, which is actually smaller than 0.63. But quickly for n equals the actual bound that you get, by the time is n is four or five, it's very close to this number, one minus one over e. So totally insane result. It's very cool. So it says we can't get one bit, but we can get something like log base two of one over 0.63 bits, which is not a bit, but it's close. It's like a bit and some change. I'm sorry, it's a little bit less than a bit. You get like 0.8 bits or something. In any dimension. It's

completely insane. So this algorithm's from 1963 or something like that. Oh, this one was actually found both in Moscow and, actually, in New Jersey, as well. One of those rare things. So okay. Now let's see how the CG algorithm – and I'll give you just the complete proof of convergence right now. It's simple. It goes like this: suppose P0 lies in a ball of radius R. But X includes a ball of radius little r. And you can take X to be something like the set of suboptimal points in your other problem, but just general. Then it says suppose a bunch of points are not in capital X, and that means that the polyhedron here still covers capital X. Then that says the following: it says if the volume of the kth polyhedron – first of all, it says that little x is bigger than that ball of radius R, and so it has a volume that's at least alphan rn, where alphan is volume of unit ball in Rn.

There's some formula for that, but it's not gonna matter to us. That's volume of Pk. That's less than this, because each time you run the CG algorithm each step your volume goes down by at least a factor of 0.63. That's the worst thing that can possibly happen to you. And so it's 0.63 to the k times volume of P0. But this thing is less than the 0.63 propagates forward. This is less than or equal to alphan Rn because P0 is inside a ball of radius R. So you get this. I mean it's really this stupid. You cancel the alphas, and you divide and you find that k can be no bigger than 1.51n log two capital R over r. It's super cool. Now bisection works like this. In R there was no 1.5 here and n was gone. So that was the difference here. So this is kinda like bisection on R, except that you're not making a full one bit of progress. You're actually making – well, we could work out what it is. By the way, notice that log base two – actually n, this thing, is actually the ratio of – it's the number of bits you were required to go from your initial to your final ignorance level. So the advantages are, well, it works, it's affine invariant, number of steps proportional to dimension and log of uncertainty reduction. And the disadvantage is this, it turns out that finding the center of gravity of a polyhedron in Rn is exceedingly difficult. Far harder than solving any convex problem. And this is the sad news, generally speaking, it is considered a substantial disadvantage when one of the subroutines called in an algorithm has a complexity far higher than the original problem you're trying to – that's generally considered a disadvantage. Now I will – we're gonna quit here, but I'll say that there are actually some very cool methods. This can be made to work by computing an approximate CG. By the way, if anyone's interested in that, there's some very cool projects that can be done on that. And we'll quit here.

[End of Audio]

Duration: 76 minutes