ConvexOptimizationII-Lecture01

Instructor (Stephen Boyd):I guess we're on. I guess I'll stay in character and say welcome to 364b and now I'll explain why that's staying in character. I believe we're making history here. This is, as far as I know, the world's first tape behind. Do we know yet if that's correct? Okay. We're gonna have SCPD tell us. I have a thumbs up. So we're making history here. This is a dramatic enactment of the events of the first lecture of 364b. So let me explain a little bit of the background. The class started not televised and we were in the basement of the history building and it was great, I could say anything I wanted. It was fun, but then there was a big broadcast so we had to televise it. So the second, third and fourth lectures were televised and all the rest will be televised, but that first lecture, which was in the basement of the history corner was not televised so we're actually doing a tape behind, which means we're going back and redoing lecture one. By the way, this is just for the benefit since other people were actually here; this is just for the benefit of people watching this on the web so we have a complete set of all lectures. So if you're watching this on the web, you can thank me, but more than that, you can thank the students who actually came to be subjected to a dramatic reenactment of lecture one. Most of them skipped lunch to do it, and yes, there's more than one of them if you're guessing. If you were wondering about that. So 364b, I'll say a little bit about it. Of course, it won't be same as what happened on the first day, but it's just so that we have something there that records the first material. It's a follow along for 364A. I guess what I said on the actual first day was that it won't be televised, I can now revise that to say it is televised and will be televised and those lectures will be available on the web as well. That's how this works.

The requirements for the class are basically homework, one of which is already assigned, in fact, one is already due in real time, but anyway, we'll have homework. Whenever we think of homework, we'll assign them and we'll pipe line them so we'll assign homework before – they won't be sequential so while you're working on homework two, we'll assign homework three and they'll be small and variable amounts and things like that. There's no final exam so no more 24-hour overnight fun. I know you're disappointed. There will be a project and so let me say that's going to be a substantial portion of the course will be a project. I'll say a little bit about the projects and then move on and cover the material. The projects aren't supposed to be a full research paper or something like that. In the past, this course and then several years when I taught the original Convex Optimization course there were few enough people to be having the project. Many of them turned into papers, many turned into thesis and so on. But we have enough people in the class this time that we can't actually do that. The projects can't be 15 page things with long stories and things like that. We just can't possibly read these or help people with them so what we're shooting for in the project now is it's really toned down. You should be thinking the final artifact of the project will be something that's on the order of five pages or something like that. We're going to be very strict on the format and not accepting weird, long and unclear things. It's gonna be in our format period. The idea is you should shoot for something – even if you're interested in something that's quite complex like you do medical imaging or you're in statistics and do something or other, the problems you're interested in might be large, complicated and so and you can't really do them justice in five pages, that's fine. What you'll do for your project in this class is gonna be something like a characteriture of it, a highly simplified version. One model for that is something like the examples in the Convex Optimization Course so that's your model for a project. They're complete, simple; anybody who knows a little bit of applied math can understand it perfectly. There will be equations there that don't make any sense, but you just have to say, you know, the return is given by blah, blah. You don't have to know that. That might be a whole course to understand that formula, but the point is if you read it, you're just saying there that that's what it is. So that's the idea behind the projects and throughout the quarter, we'll have more to say about the projects. Let me think what else to say in terms of preliminaries.

What we'll do is just cover the material so we have a complete record of the transcript. I should say a little bit about the topics we're gonna do in the class. First of all I should say the class is gonna be more disorganized that 364A; 364A has a very coherent structure, it kind of follows the book and there I don't mind saying to the people who have taken 364A I know we went way fast, there's no way that anybody could actually absorb everything in that book in one quarter. My working hypothesis is that you absorbed 75 percent of it and every now and then I'll take some time to try to remind you of some of it or something like that. And we'll have more time this quarter to absorb some of those topics. So this will be disorganized. We have some lectures prepared. The lectures this time actually have notes associated with them so if you go on the website you'll find slides but you'll also find these more detailed notes that are five pages, 10 – you should read those, we wrote them and it wasn't easy, please read them. Those have more of the details and things like that. It sorts take the place of the book this quarter. So we're gonna talk non-differential optimization, that's the first section of the course so we're gonna cover some radiance, we're gonna start on that today. We'll talk about methods that work for different non-differentiatable problems, and I should say a little about that. Of course, in 364A, we dealt with lots of non-differentiable problem, things with L1 norms, L infinity norms, and it's just not any problem at all. The way you deal with those in the 364A style is you transform the problem so you have some absolute value, you introduce a variable T and you replace the absolute value with T and then you push on the list of constraints, you know, X less than T and X minus X less than T and then now you have a bigger problem, one more variable, two more inequality constraints, but that big problem, you just eliminated one absolute value. And that's our standard method. That's the approach you should use. If you can use an interior point method, by all means do. By the way, that's what CVX does so when you type in a one norm in CVX or anything else, any kind of piece wise linear thing and max them in, it's doing for you this expansion in transformation to a larger, but differentiable problem which is then solved using [inaudible] optimization method. Then in contrast, we're gonna talk about methods that handle non-differentiatable problems directly. No transformation. They deal with it, they accept the fact that there's gonna be functions with kinks in them and they just deal with them so that's what we're gonna look at.

They will have other advantages. We're gonna see, later in the class, maybe two weeks in, that they are the key to decentralized optimization methods. We'll see other advantages that they have, but basically if you have the option to solve a problem by transforming it to a smooth problem and using a cone solver, by all means, that is by far the best method to do it. Okay. So let me talk about some of the other gross topics we're gonna talk about. So we're gonna do sub-gradients, non-differentiatable optimization; decentralized optimization, t his is really fun, this is something where you would have, in the simplest case, you would have multiple processors or decision makers that are making local decisions and they will coordinate to achieve global optimality. Very cool. Some other topics that we're gonna talk about – and then it gets kind of random and spotty – other big chunks that we're gonna look at our gonna be random ones. Let me think of if I can think of some of those. We're gonna do a bunch of stuff on non-convex optimization. For example, we'll do global optimization, that's where you have a non-convex problem but you're actually getting the exact solution. At the end, you can certify it. Those methods, which you pay for in a global optimization, you pay in is time, so they can and often do run very long. But we'll look at those methods. They're actually quite straightforward so we'll look at those and we'll also look at the other methods where you keep the speed, but what you throw out is the guarantee of global optimality. So we'll look at methods like that. Particularly, we'll look at sequential convex optimization. Other things we're gonna look at will be relaxations a bit for [inaudible] problems, we'll look at some other cool things, there's these things called sums of squares methods, I'm hoping to look at those. Hopefully, we're gonna have a new lecture on model predictive control, but at that point, it degenerates into just a set of cool things that people should know about.

Oh, I forgot one more thing. One more topic is we're gonna talk about scaling convex optimization problems up to way, way big problems. Way big, so the conventional methods – these will scale to something like at 10,000 variables, 100,000, depends on the sparsity pattern. So these will work, but the question is what if something happens and you want to solve a problem with a million variables, 10 million or a 100 million and there's a whole class of methods that basically work on these – they use a lot of the standard and basic methods I might add to scientific computing so we'll look at those. So we actually will have a section of the class on solving huge problems. Here's another lecture I want to add, maybe we won't get to it, but here's what I want to add. I want to add a lecture on solving small and medium sized problems super fast, like, real time applications. I'm talking microsecond's type of things. We know what will go in that lecture, we just need to write it and have it fit in the class. So hopefully, that will work, but who knows. Oh, I should say this. For your projects, you may have to read ahead so if you're doing a problem that involves something that's non-convex, if it's small enough and you want to make a uristic for it, if it's small enough, you might want to go ahead and implement the branch, and the global optimization method and solve instances of it because then you could say something interesting. Okay. I can't think of anything else or remember anything else, but it doesn't matter for the first lecture. What we'll do mainly – and this is the important part just so we have the record is to go on and cover the material in sub-gradients. Okay. We'll start with sub-gradients. Let me just say a little about the idea first. The idea is to directly deal with non-differentiability's so we're gonna generalize the idea of a derivative and a gradients to non-differentiable problem. It's gonna turn out – and this is not uncommon, not unexpected – what's gonna happen is you want to generalize the ideas through calculus to non-differentiable problem, and it's gonna turn out that there's gonna be two generalizations of the derivative and they're gonna be different. But the cool part is there gonna be related by convex analysis. They're gonna be sort of duals of each other so that's gonna all emerge in this because that's the idea. So we're gonna start by looking at one, which is a generalization of gradient. There's another one. We'll get to that. It's the directional derivative. But we'll start with sub-gradient so let's just start with there. Then we'll cover some gradients and this idea of strong and weal sub gradient calculus. I'll talk about that. And then we'll talk about optimality conditions and we'll see how far we get. Actually, that's lie, we're not gonna see how far – I know how far we get because this is after all, a tape behind. In this case, this is not an initial value problem. It's a final value problem because I know where the lecture is gonna end. Okay. We'll start this way. If you have a differentiable function, you have this inequality. Any convex differentiable function satisfies this. What it basically says is that this is the first order approximation of F at X and it says that this first order tailor approximation – what calculus tells you is that this thing is really close to that as long as Y is close to X. Really close means close squared is what that says. But what convexity tells you is that this function is a global under estimator of F, and now the questions is what happens if F is not differentiatable?

Well, you define a sub-gradient so a vector is a sub-gradient of a function, F, which is not necessarily – and we're gonna do this, this is not necessarily convex. It'll be of most interest when the function is convex, but it doesn't have to be convex. So you say a function is a sub-gradient of F at X if the following holds; if when you form this approximation, it is an under estimator of F globally. Now, we can check a few things out. If Y is equal to X here, if Y is X then you have equality here so basically, this thing, which is an affine thing, it touches its type at the point X, but in a place like this, you actually have some range of options in what you choose for G. You have multiple sub-gradients at a point like that. Now, wherever the function is differentiable, in fact, I'm just arguing intuitively, but it happens to be true – there's actually only sub-gradient. The reason is the tension here, if it rolls any way like this, it'll actually cut into the graph and therefore it's not a global under estimator, therefore it's not a sub-gradient. So in this simple picture here at the point X1 there is exactly one sub-gradient and its nothing more than the derivative of this function at that point. It has to be. Over here at this kink point, there's actually two – there's at least two different sub-gradients, but in fact, there's a whole interval of them and it goes like this.

And in fact, the sub-gradient, at this point, is anything in between a little interval and the interval is actually from this lower derivative to the right-hand side derivative, right? The left-hand derivative and the right-hand derivative – we'll get to that later – they both exist and any number in between those two is actually a valid sub-gradient. So that's the picture. So you can put this in terms of epigraphs, you can say that a vector G is a sub-gradient, if and only if, G minus one supports the epigraph. We can say all sorts of other things. Okay. Where do sub-gradients come up? Well, in the next three or four weeks we're going to be looking at algorithms for non-differentiable convex optimization where you just deal with, directly, non-differentiability's. So sub-gradients come up there. And it also comes up in convex analysis so this is sort of the mathematics of convex optimization. Basically anywhere where gradients appear in something – actually, for that matter, in an algorithm, in anything, we'll see, often, a sub-gradient will pop in and that's it. Oh, if a function is concave, you refer to G as a super gradient of it if the hypo – that's the hypo graph – if this thing lies on top of it like that, and that's actually something like minus a sub-gradient of minus G. Some people, by the way, refer to a vector that satisfies this inequality for a concave function as a sub-gradient. I think that's really confusing but anyway. That's how that works. Okay. So let's do an example. Let's do the max of two differentiable functions. Here's one differentiable function, here's another one and now the question is – so the max looks like this. It goes down here, differentiable, as a kink and then it goes up, differentiable, again. Let's analysis what are possible sub-gradients like at this point right here. At this point, there's only one sub-gradient and it's simply the derivative of this function at that point. Notice that this is totally irrelevant, the second one. Over here, if I say let's analyze sub-gradients here, you go up here and the only thing you get is the slope there. This one is irrelevant. Now, the interesting part is right where the kink point is it turns out that you can get here. At this point, there are multiple sub-gradients and in fact, you can see that – you can actually put something there and you can rock it between the gradient of F1 and the gradient of F2, or in this case, it's in R. So it's the derivative of F1, derivative of F2. Anything in that interval is a valid sub-gradient and so you get a line segment in this case, an interval in R. That's a subset of RN and that's the note of this, this differentiable of F, and it turns out that this thing is a closed convex set in general, I mean, for any function, it can be empty. If the function is non-convex, it's clearly gonna have points where it has no sub-gradient at all and that means the sub-differential is empty. So a function, as it curves like this, and you take some point that's sort of in the shadow of whatever is inside the convex health, there's no sub-gradient there period. Now, that's a closed and convex set. That's easy to show and here are some basic facts. If a function is convex and let's just say if X is away from the boundary – you gotta go eat, right?

Student:

[Inaudible]

Instructor (Stephen Boyd):Oh, no, no, that's fine. That's fine. One of our actresses has left in the reenactment. I should say for people watching this. You don't have to worry, everything here – these are just actors and actresses. So this is just a dramatic reenactment of the events that actually occurred in the basement of the history building 10 days ago. Yeah, so we hired a bunch of actors who look better than the real students, actually. All right. Okay. You have a sub-gradient for a convex function, basically, it points away from the boundary. That's the simple way to say it.

If a function is differentiable, then the sub-differential is the singleton consisting of just the radiant and then it's the other way around. If the sub-differential has only one element in it, then F is differentiable and it's got to be the gradient. The single point is that. So here's an example. The simplest example possible is absolute value so absolute value is very simple. The sub-differential here consists of the set with a single point minus one. Sub-differential over here is single point with the value plus one. Sub-differentials are interesting only at this one kink point, at which point, any number between minus one and one is a sub-gradient. In fact, I can put something here and I can rotate this from this all the way up to there and I still am a supporting hyper plane, the epigraph and so here it's an interval and when you plot the sub-differential you actually get a beautiful picture. This is actually plotting the set this way and the sub-differential looks like this. It's here – I'm saying if you have a point here, this is one point, and the interesting thing is I've drawn the set this way and I actually get this beautiful increasing line curve. In fact, this is the way it always looks, so if you have a convex function, you're gonna get a nice curve that goes like this and every time you have a vertical segment that corresponds exactly to a kink in the function and the height of that vertical jump is exactly equal to the discontinuity in the slope. Okay. So that's the picture. Okay. So that's the absolute value. Sub-grading calculus. So there's a calculus for sub-gradients and it's very important that you have to distinguish between two and they have different uses and stuff like that. One is – depends what you want to do. A weak sub-gradient calculus is basically it's a formula for finding one sub-gradient of a function at a point. So it's a very different thing. We're gonna find out that some algorithms are gonna require you to just find one. So, basically, you'll have to implement for F, a method called F dot get A sub-gradient and then the argument would be X, and it will return A sub-gradient. It doesn't even have to deterministic. You can call it twice at the same point and the semantics of that method merely that it returns a valid sub-gradient. It doesn't even have to return the same one. And amazingly, we're gonna find that there are optimization methods that are perfectly happy with this, strangely. So that's what it is. For example for the absolute value it has to return minus one if you're negative, plus one if you're positive; if it's zero, here's what it does. It makes a system call to find out what the time is and then it returns a number between minus one and one that depends on the process ID and the time. That's what it does. Okay. I'm just saying, that's what something like this will be. Now, strong sub grading calculus is actually formulas that can actually calculate the full sub-differential so they return a different data structure.

They would actually return a set of vectors. That's what the sub-differential is. A lot of this is going to be kind of conceptual in the sense that we don't have a way to describe an arbitrary closed convex set. We just don't have such a thing. Except in special cases, this is going to be conceptual. It's not going to be computationally. And here's a claim I make. We'll see this momentarily. I'll back it up a little bit. But roughly speaking, it's this. Strong sub grading calculus can get tricky and there's tons of books on this. By the way, if any of this stuff fascinates you and it is actually really cool, but especially this, the strong sub grading calculus, very cool stuff, tons of books. My claim is something like this – if you can compute a convex function, you can usually compute a sub-gradient, so my argument would go something like this. To say that you can compute a function means that there's some graph, there's some computation graph that you would follow to compute a function, and it might something like a discipline convex programming rule set. There's composition, there's the max of a few things and there's a few rules of affine transformations. My claim is that if I have that computation graph I'm gonna know rules for pushing sub-gradients up that computation graph. So if you make a computation graph that calculates a function, it's very easy to take that [inaudible] and to have it calculate a grade. What you need is for every leaf in that tree has to be able to calculate a sub-gradient of itself. In most cases, you can actually insist the leaves be differentiable.

Now, we're gonna assume that F is convex and I'm not gonna deal with the issues of what happens when X gets close to the boundary and stuff. There's an infinite amount of information awaiting you if this is what you're interested in. So let's looking at some basic rules. The sub-differential of a function is a singleton consisting of a gradient if it's differentiable.

Here's a simple one: scaling. Sub-differential alpha F is alpha sub-differential F. Now, here, I am overloading stuff. That's a function, right, which is where the left-hand side therefore has a data structure which is a dated type. It is a convex set of vectors. Sub-differential F is a convex set of vectors and I'm overloading scalar multiplication times a set of vectors in the obvious and standard way so that's set equality here. Okay. By the way, when you see this, you can now imagine in an objective and system. For example, CVX or something, all of this stuff could very trivially be done so basically these are formulas at each computation node or how to get a sub-gradient and you do something, you evaluate a sub-gradient of all your children and then you put them together something. It depends on is that node a sum, is the node a max and so on. And you can get quite far that way. So let's do an example here. Let's do the one norm. So the one norm is actually very cool. It's function that looks like this. It's got four planes coming in, it's got a nice sharp point at zero, but it's got four creases running up the side in the one norm and each one is along the axis where one of the XI's is zero in our two. So aligned with the axis with this, you will see a crease in that thing so it's a sharp point at the bottom, four crease points running up at 45 degrees away from the origin. Okay. Now, if you evaluate the sub-differential at any place, randomly, it's probably gonna be differential able there and the derivative is just a sign, a pattern, if X is positive that component is plus one or minus one. It's easy. It will look something like that. If you evaluate this right on the crease – so if you get a crease like this then the sub-gradient is going to be a line segment like this. And if you evaluate at the origin you get this. You get the unit vault in L infinity norm. By the way, this is not an accident. The sub-differential of a norm at the origin is always exactly – and this is just by definition – if the unit ball of the dual norm, so here, the L1 norm, dual norm is L infinity, the sub-differential is L infinity unit ball. I want to point something out and it's gonna be fun for later. You should think of the sub-differential as the size and this is just a very vague idea, but it'll be made kind of precise later – the size of the sub-differential you should associate with the amount, the non-differentiable of the function at that point.

The kinkiest point is at the origin where it's sharpest so that's kind of the idea and there it's because you can rotate it a lot of directions like this. Okay. So you should be thinking of this this way and this is, like, dual cones, which you remember from 364A, right? When a dual cone is big, fat and blunt, right, it means it's almost like a half space. And what that means is the dual cone is tiny and sharp because you can't wiggle out of a hyper plane too much. So basically, cone big, one, dual cone, sharp. Okay. So the opposite is true, too. If you have a cone, which is super sharp, it comes down to a little fine needle edge. It means when you put a supporting hyper plane down there that's tons of freedom in supporting hyper planes and that means you can have a big sub-gradient, sub-differential. You don't need to know of this. This is just so you have a better picture for what a sub-gradient is. Okay. Now, we get something sophisticated. Yeah?

Student: For [inaudible] you said we choose one of the functions that obtains the maximum?

Instructor (Stephen Boyd):Right.

Student:[Inaudible] sub-gradient methods [inaudible] – what if I choose a function that I notice within two or three percent of the maximum, would that work pretty well or –

Instructor (Stephen Boyd):Yes, it will. Well, there's a name for that. That's called an epsilon sub-gradient and there are methods based on that called epsilon sub-gradient methods and stuff. Yeah. And that's from our friends in Moscow. Oh, I should say something about this. The history of this material is a mathematical topic that's a 100 years old. In the context of actually algorithms to solve this and actually using sub-gradients, it traces back to Moscow and Kiev in the 60s, maybe 50s even. So point wise to [inaudible]. It works like this. Now, I have an arbitrath family of function that I'm taking the supremum of point wise and that gives me my function. Now, here's how you – here, in fact, I'm not gonna give the strong calculus. I'll give you a weak calculus. Weak calculus is really stupid and it goes like this; find a value that actually achieves the maximum and return that sub-differential. Strong calculus basically says find all the ones that do this, take the union of these and take the convex health. You have to take the closure. In fact, this equation is false in general. You actually need some conditions. For example, the supremum might not be achieved, in which case, you'd have epsilon sub-gradient and things like that flying all around. So if the supremum is not achieved, then this formula is wrong. In particular, the left-hand side is empty and the right-hand side is not so this is where it starts getting tricky.

Student: Why do you need the closure in this case?

Instructor (Stephen Boyd):Let's see. Why do you need the closure? Well, it's not gonna hurt because, as a general fact, the sub-differential of a convex function, at any point, is going to be closed. So it doesn't hurt. But in fact, there's simple examples where, here, if you take the convex hull of all these things, it's open. So you have to do that there. So let's do an example. Is the maximum eigenvalue of a symmetric matrix is an affine function of a variable, so non-differentiable function, we know that. Let me say a little bit about that, the maximum eigenvalue. Here's how you get the maximum eigenvalue. It's complicated. You take the matrix and you form the characteristic polynomial. You do that by writing debt SI minus A. Now, symbolically, you don't want to see that if A is more than four by four. Trust me. So if A is 10 x 10, you can't even see it. It's, like, a long book or more. Okay. You collect the terms and you now have a polynomial of degree end. You might know and might not know that. There's no analytic formula with roots and things like that. Okay. That's a famous result. That, by the way, has no bearing whatsoever. It doesn't stop people from computing roots of polynomials of degree five or anything like that. But the point is, there's no secret formula like the quadratic formula for fifth order and higher. Okay. And then when you calculate the roots, you find the largest root. Okay. All I'm trying to tell you is although, for you, it's not a big deal to talk about lambda max, it's not a big. I'm just trying to convince you. This is not a simple function of a matrix. It’s very complicated. Now, the fact is that when this eigenvalue here is isolated, so when there’s only one eigenvalue at the top, that function is actually analytic. It’s differentiable as it could possibly be because it satisfies debt, lambda, max, I minus A equals zero and actually by the implicit function theorem, that’s all analytic.

It turns out lambda is now an analytic function, locally, an analytic function of A, and therefore of A. So no problem calculating it there. But of course, if there’s ties, it gets complicated and you’re gonna get kinks and all that kind of stuff so okay. Here’s the method. This is nothing but a supremum of this function here is affine in X for each Y so here’s how you find a sub-gradient of the maximum eigenvalue. You do this. You form the matrix A of X. You then find its maximum eigenvalue and you find any eigenvalue vector associated with the maximum eigenvalue. That maximum eigenvalue is unique. If it’s a unit vector you want, it’s two because you could return Q or what do I call it – Q, V, it doesn’t matter, but Y. So Y. It could be Y or minus Y because if Y is an eigenvector so it’s minus Y and the unit vector. Fortunately, this doesn’t change. So then I look at this function here and evaluate it for Y. This thing is affine and the co-efficients are these so that’s the gradient and so that’s how you get it. It’s an eigenvalue value computation. So that’s a simple thing. We’ll look at a couple of others. Expectation – suppose you have a function of F of X, which is the expected value of a family of convex functions that depend on some random variable U.

Now, this function, by the way, is convex because expectation is gonna preserves convexity. What you need is this. For each possible value, this random variable, you need a selection of a sub-gradient like that, and then it turns out if you take the expected value of this sub-gradient that’s gonna be in the sub-differential of X. The way you might do this practically would be something like this and we’ll talk about this later in the course. Actually, I can say, if I break character, we talked about it this morning, but I guess I’m not suppose to break character. All right. So the way you would actually evaluate this and practice would be this. What you do is you generate K samples of this random variable. You’d evaluate this and you’d sum over K. Now, if capital K gets big enough, you’d argue that this, which is actually now, technically, a random variable, its variances are going to zero. It’s going to zero, like, one over K is the variance of that. So then you want to get bounds and things like that on this. Here all you do is you pick these, you calculate a sub-gradient and you average the sub-gradient and there’s more on that later, which actually was this morning. Minimization. What we’re doing is we’re just doing the calculus of sub-gradients here. So minimization is this. Recall that if you have this convex problem here, and in fact, to make it simple, we just changed the right-hand sides here. You can think of YI as a resource assignment. So you minimize the cost of operating something. You have M resources allocated, by the way, if you allocate too few, this becomes infeasible and this minimum cost is plus. The point is that it’s very easy to show, it’s standard, that this function – the optimal cost is a convex function of the resources of the Ys. Okay.

So the question is how do you calculate a sub-gradient? Okay. Well, the way to do it is this. It comes straight from something we’ve already done. You solve this problem and you find an X star, but also find, and this is what you need, an optimal dual variable lambda star associated with these inequalities so you take on optimal dual variables. We have a basic formula that says this. The G of Z is bigger than G of Y minus this, that's a basic global inequality from duality and what that says if you turn it around, is it says that minus lambda star is a sub-gradient of G at Y. This is a weak rule. It looks like this. I want to find a sub-gradient of F, which is a big composition function here, H is convex and non-decreasing and these are all convex. Here's what I do. I find a sub-gradient of the parent function, H, the calling function, and that's evaluated as point. Okay. And then I find a sub-gradient of each of the children, the argument functions, at X. Then it turns out I simply form this linear combination here and that's a sub-differential. That's in the sub-differential of F of X. It's a valid sub-gradient. By the way, this formula is the same. It reduces exactly to the standard formula for differentiable H and G for the chain rule. I don't think I will go through the proof here, but you have to argue each step here and so on. It's not that hard to do and you will have to use the non-decreasing argument here and the fact that H is convex. FI convex does have to come in; otherwise, F itself is not convex and so on. Okay. Let's look at sub-gradients and sub-level sets. If you have a sub-gradient at a point so here it is. Here is a level set for a convex function. That's this curve here and this is the sub level set inside here. At this point, it's differentiable, and therefore, the only sub-gradient is the gradient and the gradient points in the direction of maximum local increase of the function and our basic convex and equality tells us something very interesting. It says, basically, any point out here has a function value larger than at F point. That's that basic inequality. F of Y is bigger than F of X plus [inaudible] F of X transposed Y minus X. Right. That's what this says. By the way, that's very interesting. If you were trying to minimize this function, it actually tells you something very interesting and I want you to keep it in your mind for later conceptual use.

So here it is. Let's see. If I wanted to minimize X, and that's my trial point, and I evaluate the gradient like this, then this entire half space I can now rule out. I don't even have to look there because without even checking, every point here has a function value larger than there. And therefore, my search can be confined to that half space. So, roughly speaking, you should have the following idea in your head. If you evaluate a sub-gradient of a convex function at a point, a vector will come back and it'll point in that direction. Basically, what it says then is if you make a hyper plane with this being the normal, so this is the normal to the hyper plane, it says that half space, you do not ever have to look in again, ever. If you want to minimize that function because every point there has a function value bigger than or equal at your current point so I cannot be better. It says, basically, you can concentrate your search on the other half space. And now, I'm stretching things very far here. Very far in my information theory that my colleagues would not approve. I'm going to say this. You evaluate a sub-gradient – very roughly speaking, you get one bit of information about where the solution lies. You're smiling. I know, it's actually not – don't push this too far because it's totally wrong. And my argument goes something like this. If you know nothing and you evaluate a sub-gradient, you get a half space. You've divided the volume where it is, like, roughly, in half. It's a very important thing to know though if you ever wanted to know what's the value. If you evaluate a gradient or a sub-gradient of a convex function, you just learned some information. Okay. Back to the sub-level set. In the case of a point where it's non-differentiable, there are multiple sub-gradients and every single one of them provides the same thing. It provides you a valid half space where the function value is bigger than there. Something like that. So that's the picture for a sub-gradient and level sets. This idea is gonna come up later in the class and we'll say more about it. So you get supporting hyper planes to the sub-level set. I'll cover this briefly. Otherwise, we're violating the causality and then we'll quit, so I'll say a little bit about this. For a quasi-convex function you have the idea of a quasi-gradient and a quasi-gradient that looks like this. It says, if you transpose Y minus X is bigger or equal to zero and F of X is bigger, F of Y is bigger than F of X.

That's certainly true for a sub-gradient and it simply generalizes the idea so a quasi-gradient, again, the correct idea is something like this. If you evaluate a quasi-gradient, you actually eliminate from consideration an entire half space. That's sort of the idea. I should also issue a small warning here. Warning, invitation, whatever you like. If you go to Google, Wikipedia, whatever and you start typing in things like sub-gradient, quasi-gradients; you'll find an entire industry has been built up in the last 40 years, tons of papers, lots of different things. A pseudo gradient, if its non-zero, is the same as a quasi-gradient and if it's a sub-gradient, then it's a quasi-gradient, but not necessarily a pseudo gradient and you'll see all these kinds of things. Just a little bit of a warning if you do type these things in. You'll find things that are very complicated. So we're only gonna look at two ideas; sub-gradient and quasi-gradient. Okay. Now, at this point, I have caught up to where, in fact, we started lecture two so if I continue, I will violate some terrible law of causality and I hope that going back to the past and re-doing this, I haven't changed anything in the future or anything like that. Things are cool. I want to thank the, more than handful, of people who actually skipped lunch to come and make history at, what we believe is the world's first tape behind lecture. Now, there's always been times when I've wanted to tape lectures behind, but that's another story. Okay. Thanks for coming and we'll quit here.

[End of Audio]

Duration: 62 minutes

ConvexOptimizationII-Lecture02

Instructor (Stephen Boyd):Hey, I think this means we're on. Are we on? We're on, okay. So well, the first announcement I guess is kind of obvious, but I guess if you're here. We've moved. It was a challenge actually getting the word out, because the registrar hasn’t yet populated the EE364b all list.

No, sorry, it has populated it. It has just me and the TAs on it, so. In fact, let me actually start by introducing the TAs. You should know them. Jacob and [inaudible]. So familiar faces, maybe, from 364a.

Okay, so the obvious first announcement, kind of obvious, is yes, we're now on SCPD. Now that's too bad in a way, because I'd just gotten used to, well, the idea of teaching in the basement of the history corner and saying basically whatever I want and not worrying about it.

So now, unfortunately, since I guess these'll end up – these will be posted, you know, just on the web and open to anybody, I'll have to behave a little bit better. But we'll just try to pretend – I don't know, we'll just try to pretend these aren’t going to be posted, and maybe if I say anything too terrible or whatever, we'll go in and fix it before it's posted, or something like that.

So okay. And by the way, please do not – or no, please? Let's see – no, that's too polite a word. How about this: do not take that as license to sleep in. We don't sleep in, so we don't see why you should. So our requirement of attendance, we still have – that's still enforced, even though we are now going to be televised.

Okay, let's see – and a couple other announcements. I presume people will be coming, wandering in in the next ten minutes; that's perfectly okay because if you're not on this – if you weren’t actually registered or on the access list or something like that, there'd be – you'd be scratching your head reading this sign over at the other room about right now, and I'd say about ten minutes later we'll have a bunch of people coming in.

So I'll just cover some basic stuff over the next ten minutes. The first is – let's see, we posted homework one, but right before class I thought of another problem, so we have to write the solution for it to make sure it actually – it's a doable problem. And if it turns out it's actually doable, then we'll repost it and there'll be three homework – I mean, these are small things, right? These are just so you don't just watch me do subgradients, you actually get to do some subgradients yourself.

So you'll hear more about that. Let's see, the other thing is we posted a resources page. It's very crude right now on the course website, and that's something you should look at for sure. So you should be thinking about projects. I mean, you don't have to be spending all waking moments thinking about it, but you should think about it a little bit. I've already spoken to some people about it; that's good.

The resources page basically says kind of what we – you know, the first order, here are the first order of places you should look. So, you know, and it lists the obvious things, but just to be complete. I mean, definitely anything in 364a, the convex optimization book, and we've listed some other places – Google.

I mean, these are – don't come and say "I'm thinking of doing X." First – I mean, just because of the size of the class and we're doing projects, just to make it manageable, just do a little – I mean, please go and Google everything first and get a couple of papers, even though they're going to be terrible. At least then you know there's some papers on something before you come and talk to us.

We'll set up sections next week. They're not going to be sections in the normal sense, so they're not going to be, like, you know, here's a hint on homework one or something like that, because that's not what this class is supposed to be about. What they are gonna be is actually maybe just kind of – like the sections I think I'll do them for a while, or we'll all do them, and they'll just be discussions of potential projects.

The idea of a project and so on, especially early on, because – and that's something that everyone should probably come to, and we'll just – people will suggest projects and we'll just discuss sort of and aim to push the projects into a form that makes sense for the class and so on, and that would be something. And if it brings up some cool topic, that might be useful for everybody.

Oh, there is one thing about the projects. You are going to have to – you may have to read ahead. So, you know, obviously there's a small causality issue. The causality issue is this: it may be that your project will depend critically on material that we'll cover in this class the seventh week, or eighth. Or maybe we'll never get to it at all, or who knows? Or something will happen. Because it's going to be – the whole idea [inaudible] is going to be a little bit less organized and so on, be more fun.

That means, you know, you may have to go ahead and read ahead, and in fact that's why I guess someone was asking about the notes being – so we have posted a bunch of notes. Yes, those are the notes from a year and a half ago, which was the first time this class was taught. I think they're okay. You know, we reserve the right to update them, improve them, and stuff like that.

But they're there precisely so you can read ahead. So for example, if your interest is going to be something involving global optimization or if it's going to heavily involve distributed optimization, if you want to make some distributed thing or something like that, or ultra-large scale, you're going to have to read ahead. So that's kind of obvious, but I thought I'd just mention it.

Okay, any questions? I guess we probably have a few minutes before our second wave of people arrive. No? Okay.

Then I guess we'll just start in. It's going to be interesting – well, okay. I guess I should have a laser pointer. I'll bring one next time, or something like that, so I can point to various things.

Okay, so what we're gonna start today from the idea – so last lecture we looked at the idea of a subgradient, which is essentially, it's a generalization of the idea of a gradient, but for a non-differentiable problem. Actually, there is another generalization of a derivative to a non-differentiable problem. We're going to see that other definition soon, and in fact they're going to be duals, in a certain sense, so we're gonna see how that works.

So subgradient is one. It turns out it's actually the useful one. The useful one is the subgradient one – useful in terms of many things, like for example if you want to come up with simple algorithms to optimize things, subgradients are very, very useful.

Okay, so last time we got as far as this, we said that – we observed that if you have a subgradient of a convex function at a point, that's G is a subgradient of F at X. Now of course there can be many subgradients, but if you have any subgradient, what it tells you is this: it gives you essentially – I mean, here's a point – I can actually reach this with a laser pointer – so here's a point where there is only one subgradient, because it's differentiable here, and it's the gradient, and it points this way.

And what this tells you is that this entire half-space here, this whole half-space here has a function value of higher than the value at this point X1. And that means, for example, if you wanted to minimize that function F, there would be absolutely to consider this half-space.

So another way to say it is a subgradient rules out an entire half-space from consideration. So that's what a subgradient is. And by the way, that's true if the function is convex or not. It just happens only to be useful in the case when the function is convex, because it's actually true – I mean, this is true even if the function is not convex.

The problem is you'll never know. You'll never find – there's no way to find the subgradient and so on and so forth. There's no way even to know that something is a subgradient unless the function is convex.

Okay, so let's look at a useful generalization of this occurs when you have a function that is quasi-convex. So remember what quasi-convex means, it means that you have – quasi-convex means that the sublevel sets are convex, and you would typically, if you had an objective like this, you would solve it by some kind of bisection, for example.

You would ask for – if you can attain a level, that's a convex feasibility problem, and then you might do bisection, for example.

Okay, so quasi-gradient is a vector that satisfies the following. It says that if X transpose Y minus X is bigger than or equal to zero, then F of Y is bigger than F of X, so that's a quasi-gradient. I should warn you that if you do look in the literature, you will find tons of variations on these. You'll have, like, a strong quasi-gradient, a pseudo-gradient, a this and a that and that, so just watch out if you look in the literature or anything like that.

And so the basic idea is this: it says that if you have a quasi-gradient, something like here, it basically says that everything on one – it rules out that half-space. It says that everything in that half-space has a higher than or equal function value from the other one. So that's quasi-gradient. So obviously if the function's convex, a subgradient is a quasi-gradient, because that's where the whole idea comes from.

So let's look at some examples. The canonical example of a quasi-gradient function is linear fractional. So we look at a linear fractional function on the domain – the domain is going to be the open half-space where the denominator is positive, and so here if you work out – I mean, you can just easily work out what a quasi-gradient is; it's just a quick calculation.

But for example here you calculate one this way. It's just A minus F of X, zero, C. I mean, this is easy, you simply write F of X is less than F of X0, or F of X is bigger than X0, and actually that's a half-space anyway. So that's a quasi-gradient, and to check, all you have to do is check, is this, is if you say you have A transpose X minus X0, if I multiply this thing out here, that's bigger than F of X0 times C transpose X minus X0, and that tells you if you simply divide here and add D to both sides here, the X and the X0, that says F of X is bigger than F of X0, okay? So that's an example.

As another example, here's a quasi-convex function, and it is the degree of a polynomial as a function of the coefficients, right? So it's just interesting because integer value than – so that's what it – and it's kind of pedantic, even to call the degree quasi-convex, although it's not a bad thing to do.

Because the way you solve it is really silly. If you had some convex problem involving a polynomial like this, you would – and you wanted to say, for example, minimize the degree of the polynomial that satisfies something – for example, find me the lowest-order polynomial that actually satisfies some interpolation conditions, or approximate something well enough.

It's quite simple. For each fixed degree, it would be a convex feasibility problem to check if such an approximation exists. And then you could just do it bisection. You take N, you take zero, and then you take N over two and round up or down – I mean, your choice, it won't make any difference – and run bisection.

Okay, so here, what you need is a quasi-gradient, and here you could just work out what it is. It turns out you can just take the sign of the K plus first coefficient in the K plus first position, and you can actually just check, that's quite straightforward, because you get G transpose B minus A is, well, G transpose just picks out the K component and multiplies it by AK plus one, the sign of AK plus one. So this will do the trick, and of course this will imply that BK plus one is not zero, and that will imply, if BK plus one is not zero, that implies that the degree of B is at least A plus one, which is bigger than K or whatever, so that's how that works. So these are examples, and these are not hard. I mean, you just – these are, like, little exercises.

Okay, now we'll look at optimality conditions. So – yeah, by the way, I want to point out we sort of – it is important to understand. So far we've done nothing but analysis, we've had – so it's been just, it's math. But it's even worse, it's kind of circular math where you just introduce a bunch of definitions and then do things but never actually say anything that helps you or whatever. So that's a current state.

That'll change, actually, later this lecture, but for now that's the current status. Actually, the first inklings of something – that you've actually done something will occur now, and they just sneak up on you very quietly, so they're really quite weird. So here it is.

You have a differentiable function, the condition is for optimality – necessary and sufficient is that the gradient should vanish, that's it. And, you know, one way it's very easy to show. If the gradient vanishes, you write down this inequality that says – let's see if this is gonna work.

If a function is – who's going to be able to [inaudible] through the sight line very carefully here? Let's try this. So we have this, and the question would be, can anyone see that? Okay, and that's good enough. So here's your basic inequality. Obviously if the gradient is zero, it basically says that for all Y, F of Y is bigger than F of X, but that's basically the definition of X being the global minimizer of F, right? So that's simple enough.

And then the other direction is also easy to show as well. In fact, the other direction you can show lots of ways. One would be to say suppose the gradient isn't zero? If you took a small step in the direction of the negative gradient, the function would go down, okay, which would kind of cast some serious doubt on your assertion that X was optimal.

I mean, what I just said was actually a proof, but just, you know, in words. Okay, all right.

So non-differentiable, I mean, it's really simple. It looks totally innocent, and it's this: a non-differentiable function is minimized by X, star if and only if – now here the sub-differential is a set, so, and if and only if zero is in the subgradient. So that's the condition.

One way is completely trivial to prove. It's the same thing, right? Because in fact here's what a subgradient is. The definition of a subgradient is this. The definition of a subgradient is that for all Y, this holds. But if zero is a subgradient, it says that for all Y, F of Y is bigger than or equal to F of X. So one direction here is just a tautology.

I mean, the other direction is just as easy as well. In fact, it goes the other way. Suppose I told you X is the minimizer of F? Global minimizer. By definition, that means that for all

Y, F of Y is bigger than F of X. That's what it means to be the global minimizer. Therefore it's for sure true that this inequality holds when G equals zero. That means zero is the subgradient, okay?

So, now when you first see this you think, nothing happened. That was completely trivial, that was circular, there's nothing there. So in other words, not only is this true but it is completely trivial.

Okay, that's sort of it. By the way, this case is an interesting one, right, and it shows you – it says that it does not say that zero is the only subgradient. And in fact in this case there are many subgradients. What you can say about the subgradients here is they straddle zero, is what happens here, obviously, right? You have negative slopes and positive slopes, but zero is in the subgradient.

One comment: weak subgradient calculus will not help you with this. I mean, remember weak subgradient calculus, I mean, if you want to think about it operationally, is this: you have a method that will return a subgradient at a point. And then point here is I could easily return for this function. At X0, I could return the subgradient minus point one. And it is a subgradient, there's absolutely no doubt about it. It is not zero, right?

So weak – basically what that tells you is that stopping criteria based on weak subgradient mechanisms are not gonna be simple. Now if you have strong subgradient conditions you can do that, because you have something that somehow abstractly in some way returns the entire subgradient, and of course you have to have the containment question. Can you check if zero is in the subgradient or something like that? Actually, we'll see there's a very good way to check that.

Okay, so what's weird about it is that although it seems like we've done absolutely nothing, it turns out we actually have done something. It's kind of weird, but let's try it. So let's take these piecewise linear minimizations, you want to minimize piecewise linear function. And according to this trivial analysis, it says zero – the optimality [inaudible] zero is in the sub-differential of F at X star. But we know what that is, because this is the max of a finite number of functions, and that's equal to this.

It's the convex hull of the AIs where I are the active constraints, okay? So that's what the – and it's a polyhedron, that thing. Now the convex hull is just – I mean, another way to say it is just you're basically saying this: there are lambdas which are positive lambdas that add up to one, so it's like a probability distribution, and the expected value of the As is zero, okay?

Oh, and I should say this condition. In fact, this should look very familiar, it's a complementarity condition, right? Because it basically says these lambdas here – I assigned a lambda for everybody, but up here, that's the convex hull only of As corresponding to active ones, and therefore I have to make this thing mean the same as that. I have to say the lambdas are zero if you're non-active. This is what it means to be non-active. Yeah?

Student:[Inaudible] do that again?

Instructor (Stephen Boyd):Yeah, we did that last time. It was – yeah, we looked at – no, it shouldn't be obvious, but it's true, it's not hard to show, but it's certainly – it's actually not hard to show at all in this case. I think we looked at that last time, so that was our strong subgradient formula, right? It said that if you have the max of a bunch of things and you wanna calculate the sub-differential, you would calculate this way.

In the max you'd look at – first you'd single out every one who's active, and you'd screen for the active guys in the max, and then for each of those you'd ask them to produce a sub-differential at that point. Then you take the union of those and the convex hull, and that's the sub-differential of the max, the parent in the tree.

Okay, okay. Now this looks pretty familiar because they're exactly the KKT conditions for this LP formulation. So that's the epigraph form, and I should add, you know, this is how you would solve this problem. I mean, anybody here, this is how you, if you had to solve piecewise linear minimization, you would do it by basically forming that.

Or, for example, if you use CVX or one of these other high-level things, that would transform that for you into this top thing, this top LP, and then solve it. But whether you know it or not, you'd be solving that problem up there. And the optimality – the dual of that is this thing right here, it's maximized B transpose lambda, subject to lambda positive, and A transpose lambda is one, and one transpose lambda is one, and that means lambda's really a probability distribution.

And this should fit well with all sorts of things. For example, suppose you minimize a piecewise linear function in – it doesn’t matter, R2. Let's minimize it in R2. And suppose when you find the optimal X, what you allege to be the optimal X, suppose only one of the functions is active there. What does it mean? First of all, could it happen?

Yeah. It means it can only happen if the slope is zero, and that'd be a piecewise linear function that looks like this, right? It only looks like that. But more generically, it's going to take a couple, and that means you're going to be at a point where you're going to be at a sharp point. So that's how that works. Okay.

Okay, let's do the constrained case. I mean, it's beautiful because it's sort of the generalization, it's the prettiest generalization of the KKT conditions. So if you optimality – I mean, I could add the equality constraints in, that's absolutely no different. It just would clutter the slide. So here, this is the constraint case, but now F0 and FI are non-differentiable. They are not differentiable, so you can't even talk about the KKT conditions there.

So these are the generalizations of KKT conditions, and they basically are the following. Obvious, if X is primal optimal, it's got to be feasible; that's a requirement for being optimal. The lambda Is have to be bigger than or equal to zero, that's obvious. And this is the condition here, it's exactly the condition that would be gradient, and gradient if these were differentiable.

In the case where they're gradient and gradient, that would be equals because the right-hand side would be a vector, not a set. Here, it's a set, and of course everything here is overloaded for sets on the right. So that's what that is.

So the actually only difference, it's quite pretty if you switch out the symbol – well, you switch that to an equals, from an equals to an N, and you switch the gradient symbol to this symbol here, the sub-differential symbol. And you have to reinterpret everything properly.

So these are the conditions, and they're quite pretty, and it gives you a rough idea of how these things work. It's actually quite easy, I think, to show – at least one direction is easy. The other direction's not hard or something like that. Okay.

So now we're gonna cover this second idea of a derivative, so let's go back to the beginning and the issue is this: here's what you wanna do. You have a non-differentiable function and you wanna talk intelligently about a derivative of it. Well, I mean, the normal derivative doesn't exist, right? So that's – I mean, at some points it doesn't exist.

Even more than that, at points that we are unusually often interested in, it doesn't exist, right? Because if you're doing L1 or anything like this – of course if you do any kind of optimization, you're going to tend to hit these points that are corners and things like that. I mean, that's kind of what optimization does.

Okay, so one generalization is subgradient, and you've already seen kinda what it does. I mean, optimality conditions are beautiful, all sorts of other stuff comes out very nicely with subgradient.

But it might not be even the first one you'd look at. The first one you might look at would be something called a directional derivative. So the directional derivative of a function is – well, it's what it sounds like. You say the directional derivative of F at X and in the direction delta X, so this is a function of delta X, is simply the limit, as H goes down to zero from positive of X, plus F of X plus H delta X minus [inaudible].

Now if it's differentiable, that thing there converges exactly to grad F of X, transpose delta X. In other words, if F is differentiable, then F prime exists and is a linear function of delta X, okay? In fact, the linear function is grad F of X transpose delta X, okay?

Now, this function here is extremely easy to show. This thing has to be – it's homogeneous, it's always homogeneous. What it might not be is linear, and we'll see simple examples where it is not linear, but it's homogeneous.

So if it's linear, it tells you this. Now this is actually not a trivial fact, but it's – and I'm not gonna go in – I mean, there's, like, nine books and things, many books, many, many books on convex analysis that goes into the horrible details of this. But a very basic result is this, is if you have a convex function, roughly speaking – I mean, let's say and you're inside the – and you're in the interior of the domain, you're away from the boundary, then you have a directional derivative in all ways, all directions, okay?

So, I mean, that kinda makes sense. Let me go over here and draw some pictures. So if you have a function like this, right, you're not differentiable there, but you have a directional derivative. And in this case, it's homogeneous, but a homogeneous function on R, a positive homogeneous function on R, you only have to say what it is from the left and from the right, right, because you know what it is for plus one and minus one, you're okay.

And in fact, in this case, F prime of X and then one is what we would generally call I guess F prime plus of X? That's the right-hand derivative, and it's this slope. And then if you do this like this, minus one, that's the directional derivative in the direction minus one, you get in this case F minus prime of X.

Now if you're differentiable, these two things are equal and they both coincide with the derivative. In this case, there's a gap between them. And for example, this thing – well, they're both negative, but that's less negative than that, okay? Everybody see that?

But what's interesting about this is this happens in RN, not just R. So take some horrible function in RN, non-differentiable, go to some point where it's got all sorts of sick stuff there. The point is that in any direction you move in, if you just move along a ray, it will be differentiable, it will have a directional derivative. Okay? Now if that directional derivative function is linear, then it means the function is actually differentiable. That's directional derivative.

Now the question is how is it related to a subgradient, and in fact they're related by duality. So, which sort of makes sense. And I think I can – yeah, so I think we can – so here's the general formula. It says this: the directional derivative in the direction delta X is the soup of G transpose delta X where G is in the sub-differential. And so that is the – this is called the support function of the sub-differential.

So to write a very compact expression for it, you'd say that the support – so you could write something like this: F prime equals, you know, the support function sub-differential F at X. So that's just a very compact formula that tells you this.

And let's actually look at this and see if we can understand how this works. Let's do the differentiable case. So in the differentiable case, what is the sub-differential? So your F is differentiable at that point, what's the sub-differential. What?

Student:[Inaudible]

Instructor (Stephen Boyd):Single point. Okay, it's a single point. So therefore, what's the supremum of, you know, the single point? In fact, that single point is grad F of X, right? So what's the supremum of grad F of X transpose delta X? Sorry, there's no supremum because there's just one point. But then it just reduces to grad F of X transpose delta X, which is exactly what that's supposed to be, okay? So it's linear.

Okay, so that works out. And now let's try to figure out like in this case, what it is geometrically. So here's the sub-differential. By the way, when you see a big sub-differential like that, actually you should be visualizing what that function looks like at that point. Like for example, is it sharp? First of all, is it differentiable at that point?

No, because it's got a big old subgradient. Then there's clearly non-differentiable, and the question is is it a knife? So you know what I mean locally by knife? Knife is something that has a few sides that go in like this and you can cut with. Or is it a needle? These are technical terms, I'm not insane. I mean, sure, I just made them up, they didn't exist 14 seconds ago. But they make perfect sense, okay? So is it a knife? Why not? Exactly.

So a knife would correspond – a knife in the function like this, a crease – let's call it a crease. Whatever, it doesn't matter. If you had a crease, if the sub-differential gets big enough, it becomes a knife, it becomes dangerous. But let's just say it's a crease. So if it's a crease, but a one-dimensional crease like this, it means that actually in one direction it is differentiable, and therefore in that direction, the subgradient, sub-differential was small, and it would be a line, just like you said, a line segment.

This is big, and so this means this is a point. This is like a needle. So in other words at that point – oh, by the way, is that point global? Is it globally minimized? Is that point supported by a flat – is it at the bottom? How do you know?

Student:[Inaudible]

Instructor (Stephen Boyd):Zeros in, precisely. So it's a point, but it's not at the bottom of the function. It's sort of off like that. So you have a point over here. Okay.

And now you can actually calculate some things. So it says if you go in this direction – by the way, does the function increase or decrease if I step in this direction delta X? Does it increase or decrease? I'm asking what's the sine of F prime of X semicolon delta X for the one shown. Does it increase or decrease if you go in that direction? It increases.

Yeah, because – I mean, there's no scale here. So the point is that point is positive. That dashed line kinda shows you it solves that little – in this case that's an LP, because I drew this as a polyhedron. That's an LP. That shows the optimum value. It increases, okay?

Now let me ask you a question. Suppose you say okay, no problem, if I go in the direction delta X, the function will increase. You go, not a problem. Now we'll go in the direction negative X. Please tell me what happens if you go in the direction negative X. Negative delta X, sorry. Negative delta X. What happens now? Unclear? No, I think it's clear. You go in the direction –

Student: Increase, and if you go in the other direction, it'll decrease.

Instructor (Stephen Boyd):That's true, it's not guaranteed. Okay, so that's – oh, you know, damn, I drew it the wrong way. Sorry. Do you mind? Let's just visualize this thing going down to here. It is – your answer was exactly correct. It's unclear because you'd have to project a normal out – well, hm. What do you think? What do you think [inaudible]? I think you decrease if you go in that direction.

But let's imagine the subgradient went down here like that. It might have, right? So it goes down here. In this case, if you go in that direction, you can go a positive amount and that says you get this – so actually here I can make another one. How about this, if I go in this direction, you increase. Why? Because the supremum of X2 over this thing is the height.

The highest height you can go above this thing is positive. I mean, go up. If I go this way, if I increase X2 a little bit, my function goes up. If I decrease X2, my function goes what?

Student: Up.

Instructor (Stephen Boyd):It goes up, okay? So let's try that again, okay? If I increase X2 a little bit, I increase the second coordinate, the function value goes up. You say no problem, okay, so I will decrease X2 a little bit. And what happens to the function? It goes up again, okay? So my comment on that is welcome to the world of non-differentiability, right? Because if it's differentiable, then if you go in one direction – if a function is differentiable and you go in one direction and the function goes up, then you just – no problem, you turn around and you walk the other direction, it's gonna go down.

Why? Because you get a linear approximation nearby, so. And if a linear function goes up in one direction, it has to go down in the opposite direction. Everybody okay?

So in this case it means that that crease or point or whatever is such that X – and if you move X1 and X2, it's gonna – X1 and X2? Sorry, if you move X2 up or down, the function goes up. Okay.

Oh, let me ask you this, let's see. Oh, we'll get to that in a second. Okay, so we'll get to the idea of a descent direction. So a descent direction is – well, it's what it sounds like, it's a direction where if you step a small amount in that direction, the function goes down. Yeah, some – it can be less than or equal to zero or less than zero, but let's not worry about that. So that's a descent direction.

Now for a differentiable function, the gradient is always a descent – a negative gradient is always a descent function, always. You step a small enough distance in the direction of the negative gradient, and the function will go down, assuming the gradient's not zero, right?

So okay. Now for non-differentiable functions, this is the part that you have to sort of get, subgradients are not like – negative subgradients are not descent functions. Actually, this is going to be useful because later today we're gonna have algorithms that are gonna step in negative subgradient directions, and that's weird, because you're stepping in a direction where the function goes up.

You might ask why would anyone do that. Well, we'll get to that. But that's the story. So here's an example, it's this function, and this is a valid subgradient. That's a valid subgradient at that point. I mean, if you draw the – you extend the line, you'll see it's a subgradient.

Okay, now what would happen if you step in the direction negative G? You'd go in this direction, and you are pointing out, you're aiming out of the level curves. You're going uphill. So here, if you go in the direction negative G, you are going uphill. Okay? Then you say, no problem. Let's go in the direction G. How about G? Is that an ascent direction, descent direction? It's what? If you go in the direction G.

Student:[Inaudible]

Instructor (Stephen Boyd):It's an ascent direction, yeah. It's worse. But the point is that you got big trouble here. So if you step in the negative subgradient direction, it is not a descent direction, okay? So that's – and you know, this is kind of – well, anyway, so that's it. But it's something that's very important to understand, so what's gonna happen is for non-differentiable functions, the concept of a derivative bifurcates into two things, and they're kind of duals, right?

One is the directional derivative and one's the subgradient. So half the stuff you know, half of your [inaudible] about derivatives is gonna extend – let's say 48 percent of this stuff is gonna extend to the subgradient, and another 48 percent is going to extend to the directional derivative, and 4 percent will just be false in a non-differentiable case. I made those numbers up.

Okay, so then you might ask if you – is there a way to characterize the subgradient, and there is, and it's very interesting, and it's gonna be the key to understanding all of these – well, I could write down the algorithms right now, because they're one line, the algorithms we're gonna write down. They're gonna be one line, so we'll get to them later today.

I can write them down, but to actually understand it, this is the key to it. Otherwise it just looks totally bizarre, even though the proof is, like, three lines. So here it is. So what a negative subgradient is is this: a negative subgradient is a descent direction for the distance to the optimal point. Actually, the distance to any point that's better than the current point.

So let's just go back and check that here. Yeah, here, okay. So if you step in the negative subgradient direction – I'm not gonna write on it, don't worry. Oh my god, look at that.

The whole time I could have – this is great, okay, wow. Now it's not – I feel like I – oh well, anyway. All right, so this is – wow, look at that. Now I can actually see it now.

So okay, so negative G is you step in this direction. We've already discussed, if you step a small amount in the negative subgradient direction here, your function goes up, okay? Which does not sound like a winning move if you're trying to minimize the function, okay?

Or I should say if you're trying to minimize it greedily, this is not what we call a winning move, okay? However, the interesting part is that's the optimal point, the minimizer, and so what this thing says is if you step a small enough amount in this direction, your function went up but your distance to that optimal point went down, and that is actually correct, okay?

So basically we're gonna see a bunch of algorithms – boy, are they simple – and they're gonna – but the Lyapunov function – so does everyone know what a Lyapunov function is? So [inaudible] know? Okay, so in tons of fields, you wanna prove something converges or something happens or some condition holds, like in, I don't know, in computer science you wanna verify that something, you know, some condition always holds, for a dynamical system you wanna verify that a certain control system on an airplane will always – you know, where you'll never deviate more than 100 feet from your altitude or this kind of thing.

In optimization you wanna prove that something in an algorithm converges, and so people have lots of – one very general approach to this is to come up with a function that decreases at each step. It may be the function you're interested in. For example, if you're minimizing a function, it could be objective value. That'll be false here.

But generally, it'll be some – in any case, it can be any old function, right? And it's a function that's just going down. And that's the key to kind of proving something converges, okay? So that's called the Lyapunov function in control, it's called – in optimization, one name used for it, or algorithm design, it's called a merit function.

And, you know, you'd say the merit function goes down. Of course, I guess if it goes down it should be called a dismerit. Well anyway, it doesn’t matter. It goes down. And it may not be what you want. I mean, it may not be the thing you're interested in. For exactly, very often in control systems it turns out a lot of these functions, which are Lyapunov functions, actually come from, like, energy-like calculations and things like that.

Okay, so all right. So what this means is – and by the way, all the algorithms you’ve seen so far, or differentiable things – you’ve seen Newton method, you know, gradient, scale gradient, blah, blah, blah, all these things, they're all descent methods. Oh, actually, we saw one that wasn’t, we saw one that wasn't. Exactly one, and that was the [inaudible] Newton method.

But all the ones that you actually messed with were descent methods, in the sense that the merit function was the function. So at each step, the function went down. Of course, it's not enough to say that the function goes down at each step to prove convergence, except that a convergence has some value. You still have to prove that the value it converges to is the optimal one. But that's a start.

So it turns out here, what's gonna go down in algorithms based on subgradients is not the function value. The function value's gonna go up. In fact, it's gonna often go up. What's gonna go down, actually, ultimately, is the distance to the optimal set. That's what's going down. Of course, you don't know what the optimal point is or anything like that, but at least this is what's going down. Okay.

So to show this, you simply write this out and you write out the square. I write – I couple these two terms and I get that norm squared. I get then the norm squared of that guy, that's here, and then I get this cross-product, okay? Now here, this, I replace this with this, and this goes up because of my inequality here, and that's just the definition of subgradient here.

It says basically that – and G is the subgradient at X, or a subgradient at X, so that – I don't know, should I do this? Sure, why not? So it basically says – you know the following – that F of – let me get it right. Here we go. You know this is – you start from F of Y is bigger than F of X, plus G transpose, but should I have used X and Z or something? Well – yeah, well, you use this one.

And then you subtract, and I should have called that Z, maybe, or I should swap these or something like that to get it the right way. Do I want F of Z? Yeah, let me make this F of Z to make it work out nicely, like so, and then I claim that inequality shoved in here does the right thing. It does, because you just switch it around. That's F, so this says basically something like this: F of X minus F of Z is less than or equal to G transpose X minus Z.

I think that's what I want. And then that's good enough. So it says that – let's see, it says that this thing is smaller than that. It's a minus sign, so everything's cool. Okay?

All right, so – oh, and there's one more step of the proof, that's this: you choose – so actually, let's analyze the terms. I want that to be less than that, okay? That's not helping, because it's positive. But it scales by T squared. This term absolutely helped, because the assumption here is that F of Z – this is less than that. If this is the optimal point, if that's X star, this is positive.

In any case, [inaudible] positive, and so that's a minus sign, T is positive, that's positive. So this is the good term and that's the bad term. But the good term scales like T and the bad term scales like T squared. So for a small enough T, the good term overwhelms, and this thing is strictly less than that, okay? So that's the picture. Just save that as an idea.

Okay, descent directions and optimality. So if you have a convex function, finite, you know, near a point X, then two things [inaudible]. Either zero's in the sub-differential, in which case you're done, right? But it's not particularly useful unless for that particular problem you actually have a handle on a strong subgradient calculus or something. If you have a method to calculate the full sub-differential, then this tells you something. Otherwise, this is useless. Okay.

Otherwise, if you're not optimal, there's a descent direction, and that says – there's a descent direction, and in fact the steepest descent direction is this: it's the smallest point in the subgradient in the sub-differential set, okay? And we can draw – let's see if I've even drawn some – yeah. So here's a picture. So here's a sub-differential, our function is quite pointed there, right?

It's got a – it's quite pointed. And zero is not in that function, so that means there is a descent there. It means number one, you're not optimal. So then how do you find a descent direction? You find the smallest point in that sub-differential, and that happens to be that vertex shown there. I can use my new technology – okay, there.

So that's the closest point to zero. That's the smallest point in the sub-differential. And then if you go in that negative direction, that – by the way, this thing, that is a subgradient. Well, of course, because it's right here in the sub-differential. This subgradient happens to be in the actual descent direction for the function. Why? Because if you find the smallest point, just the optimality conditions for finding the smallest point, is that well, if the smallest point were zero, you'd be done.

So if it's not zero, it says if you go in the other direction, you're completely free and clear of this. And that says that there's actually a descent. And the distance here actually gives you the direction, something like the negative directional derivative, okay? So that's the picture.

By the way, let me ask you a couple of questions. Is this a descent direction? It's going in that direction. See if your – how your [inaudible].

Student: Yes.

Instructor (Stephen Boyd):It is, okay. How about – how about if you go in this direction? Okay here, here, how about that? Is that a descent direction? I think that one's – what do you think? That was, like, absolutely not. Because here, you would go out to a point there and you'd have very – the soup, the support function evaluated there is going to be quite large and positive. You're gonna increase.

That's, like, ridiculous if you go in that direction, right? That's almost like the steepest ascent direction. And then down here is – how about, you know, if I went along this axis? If I stepped along this axis, would it go down or up? You go along this axis, what do you get? No, no, it goes up, way up. You evaluate the support function. You ask yourself how far can you go in this direction in this set, and the answer is way over here, and that's way, way positive.

So in this direction you go down, weakly, in fact – in fact, it depends on this little gap here. If you go in this direction, you go up pretty steeply. How about if you go, like, in this direction, here? That a descent direction if you go like –just staying along this line here. Is that a descent or not? No, that's an ascent direction. And if you go in this direction? Ascent.

Okay, so – by the way, you couldn’t possibly have this with a differentiable function, right? I mean, there's no way you could go have an ascent direction one way, turn around, go the opposite direction, it goes up there, too. So, impossible.

Okay, so this is the picture. Actually, what this means is that there's a – this is the basis for a whole bunch of methods – by the way, not one of which we're gonna look at for minimizing non-differentiable convex functions. It goes like this. It basically says the following: at each step you get the sub-differential. Then you solve this problem.

You minimize over the sub-differential the norm. If you find that zero is in the sub-differential, which you would have to find if you actually solved this problem, then you stop and you actually have a proof of optimality, because you have zero in the sub-differential, okay?

Otherwise, you find the smallest point in the sub-differential, and you step in the direction, the negative of that direction. And that's called steepest descent for non-differentiable functions, and we're not gonna study it. It kinda works.

And if you read more about these other methods, you'll find that that's sort of a theme. The methods that are sort of based on this kind of idea that sometimes work quite well are things called bundle methods. You will definitely see that if you poke around. It won't take you too long to find those.

Okay, so now this covers all our analysis of – what is all that? [Inaudible] Well, why not? [Inaudible] Mm-mmm. Hm. Interesting. Will I be able to do this? See what happens – no, hm. Oh, let's see now, okay. How about just finding a terminal – oh, no, that's maybe [inaudible]. Sure, okay. Everything's fine, don't worry about a thing here.

Sure, okay, and that was – no, no, no. No. All right, all right, well, we'll just have to figure out what's here. Hm. Okay, better. Where is it? [Inaudible] method – slides, there we go. That gonna work? Let's see. Look at that, see? Okay, nope. Great. Yeah. There we go, okay.

All right, okay. So what we did so far is just analysis, just analysis. It's cool analysis. Some of it was not useful, I mean, basically. The stuff that requires strong subgradients like the steepest ascent stuff is not – that's not particularly useful, I think – I mean, or just it seems in practice it's not.

Subgradient methods are, so these are – we're gonna look at these. I can say a little bit of history of these, you'll never guess where these come from. Actually, you might be wrong, because some people might argue they come from Kiev. So they're either from Kiev or Moscow, that's your only two possible choices.

So you'd say both, okay? So yeah, so guess what? These methods are from the Soviet Union in the sixties, surprise, surprise. But what's so weird about these things is how simple they are, and they look really stupid. In fact, I'll tell you a story about it in a bit.

Okay, so here's a subgradient – and I'm not kidding, it really is this simple. Here's what it is, ready? You take the current point, you find a subgradient, and you step in the negative subgradient direction. Multiply it by step size. So far, that's fine. So – and this is a weak subgradient. Any subgradient – by the way, including a subgradient where the negative subgradient is not a descent direction.

And by the way, what that means is this is not a descent method. You'll run this algorithm and the function will go up at certain steps. Do you have a question? Yeah.

Student: How do you know they're subgradients? Which one would be the best [inaudible]?

Instructor (Stephen Boyd):Well, if you knew all the subgradients, you would probably do pretty well by taking the [inaudible] of the norm, which would be the steepest descent point. However, interesting thing is I think these methods are actually structurally slow – they're slow no matter what. So in fact the nice thing about these – the bad thing about these methods if they're slow.

By the way, they can be fast in theory. The bad news is that they're slow. The good news is – well, in a weird, perverse sense, the good news is they don't get much faster if you're much smarter and try to calculate fancy subgradients and things like that. They go from being, like, terribly slow to quite slow. Now let me explain why that's good news.

That's good news because that's just work you don't have to do. You don't have to write some – I mean, you can be totally sloppy, right? I mean, I'll give you an example. Suppose you're doing minimax and you have the maximum of a thousand functions you wanna minimize. That comes up all the time, okay? So you could do this, you could evaluate – you know, okay, to evaluate the max, you have to evaluate all those functions, right? You have to evaluate a thousand functions, calculate the maximum of those.

Now, typically some will be tied. If there's no ties, no problem, right? But there'll be some ties. If you keep track of the ties and calculate that convex hull and solve a QP to get that best one, you could say – you know, [inaudible] look in the code, you'd say look, I'm being super-smart, I'm getting the best subgradient and all that, and this'll work slightly better. So you went from – anyway, so it's just not gonna have a big effect.

Okay. All right, so as I said, it's not a descent method. So because it's not a descent method, in a descent method if someone stops you early and says, what's the best point you’ve got so far? It's the current point, right? Because you make progress every time in a descent method, so there's no – [inaudible] another concept of the – you only have the concept of the current point.

So with a non-descent method, though, we'll encounter several throughout the class, actually, you need to keep a pointer to the best you've found so far. You just – you know, if it gets worse, you keep going but you save somewhere the best point you found so far. Okay, and we'll call FK the best point found so far.

Okay, so here's some step size rules, and let me remind you how this – when you first see this, you say hey, that's just the gradient method generalization to non-differentiable functions. Let me assure you, it is not. It just looks like it. So – in fact, I can even tell you a story about it.

So the first time I found out about these methods I was a grad student, I was reading, and I got some Russian paper on something and at that time, I guess still now, there's something called Automation and Remote, or one of these things that was translated basically by a company funded by the CIA translated all the Russian papers badly. But I got very used to Russian typesetting, because they didn't – they just kept the typesetting the same, but the English was written by somebody at the RAND Corporation, something like that.

So I conjectured that they didn't know what they were doing, and the original Russian, although I haven't verified this, had subgradient, but it got translated as gradient. So I'm reading this paper, I'm on, like, page two, and it says we'll just use a gradient method.

So we'll just take a gradient – and I'm looking at it, I'm like but just a minute, that function is non-differentiable. There is no gradient. But then they looked just like that, and I said well, that's the gradient method, for sure. This function's non-differentiable. There's no way. This can't – and then I thought – so then I had to go cool off and thought okay, either this guy is a complete – has no idea what he's doing, I mean, at all – I mean, this is the most fundamental thing – or there's something very interesting here.

So it turned out to be the latter. So it was the subgradient method, badly translated as gradient. So you'll still find that. By the way, if you find Russian literature, you will actually find people referring to gradient, meaning subgradient. So you have to be very careful.

Okay, all right. All right, so this is gonna be – now in the gradient method that you know and love for differentiable functions, it works like this: you calculate – this is the gradient, and you calculate alpha K lots of ways. One way would be an exact line search, so you'd find out the best point.

You could do a backtracking line search, start with alpha equals one and times equals beta until you got some sufficient decrease. Everybody knows what I'm talking about, right? Okay. So you would chose alpha and the step length in a – I mean, you would never choose it ahead of time. You'd never say that my 14th step size is going to be point two.

You'd wait until you got to the 14th iterate and try some step sizes, and if point two was the right thing, that's what you would do, right? I mean, that's it? Okay. So here are the step sizes for the subgradient method. I mean, it's just – the whole thing is looking really implausible, if you ask me. I mean, it just – you know, we start with the fact that you're gonna move in a non-descent direction when you wanna minimize a function, and the next part that really puts it over the top is this.

Is that for these methods, you basically set the step sizes ahead of time. In other words, you don't even adapt them. I mean, there are fancier methods where you adapt what the step size, depending on where you are. But basically, these methods work fine if you just say the step sizes are – for example, here, ready for some step sizes? One over K. So in the [inaudible] iteration, your step size will be one over K.

Oh by the way, the function has some kind of domain issue, right? So you can't do one over K. You have to modify that. But the point is here's one, just one over K, here you go. So the whole thing's ridiculous. I mean, here's some step sizes, all very implausible. You have a constant step size, another one, here's a constant step length. That means that you get the subgradient here, you divide the step length, and that means that your actual – the distance you step is exactly gamma here, okay? Even more – these are just, like, if you think – these are just ridiculous. Highly, highly implausible rules for step length.

Another one that's gonna come up is this, is step lengths which are square-summable but not summable. And so this would be, for example, one over K would be one. And then you have non-summable diminishing and this would be one where the limit of these things go to zero, but they're non-summable.

Obviously, this implies that, so. So this is the weakest. And the classic example here is one over square root K, and why that is will come up in a minute.

So I just wanna emphasize how silly this algorithm's looking. So it goes like this, it says you're at a point X and you call get subgrad at X, so you – and this thing returns. It has many choices of subgradients, I mean, if it's non-differentiable, and it returns one. And there's no restrictions on which one it returns.

In fact, multiple calls at the same point could return different subgradients. It makes no difference. Doesn't even have to be deterministic. Just returns the subgradient.

Okay. Which need not be – the negative of that need not be a descent direction. Could be an ascent direction. And in fact it could be even worse, because your alpha's gonna be positive. But in any case, it could be even a direction where either direction you step, the function goes up, okay?

Then you'd say well that's okay, because there'll be a sophisticated line search to take care of that. No, the line search is basically predetermined. It's just fixed, it's like point one, that's the line step.

So, I mean, I don't know about you, but this is not sounding like super – I mean, it doesn't seem that promising when you first encounter it.

Okay, so there's a couple of assumptions here. We'll just say let's assume that its actually [inaudible] below and there's an optimal X. And then we'll take a Lipschitz constant on the function F, and that's the same as saying that the subgradients are bounded by G – I mean, roughly.

And then the other thing is we'll take R to be the distance of our first iterate to the optimal point, so capital R. So actually capital R, especially if it's inequality, is something like it's a distance measure, it's measured in meters, if X were in meters, it's a distance measure of our ignorance, because it's how far.

Before you start the problem, if someone has given you an X one and an R and a G, and if someone says how far are you from the optimum, you have to say R. By the way, how far are you from optimum in terms of the function itself? You've done nothing. You've only been given the following data: X one, R, and G. How far off is F of X one from F of X star? It has an answer. RG, thank you.

So RG, R is your measure in meters, if X is in meters, of your ignorance. I mean, before you even start it, if your prior ignorance. RG is your ignorance in F star. Because if someone says you have F of X one, and someone says well, how suboptimal are you? You'd have to say I could be up to RG suboptimal, right? So actually just bear that in mind, because RG is gonna come up.

Okay. Oh, by the way, I should say that the proofs I'm gonna give here are super-simplified. You can go read the Russian ones if you like. They're actually a couple of pages longer, but they do away with, for example, this one, okay? So you don't need this. This is just because I like groups that fit on a half a page or a quarter of a page or something like that. But anyway, I think this is the main idea.

Okay, so here's some convergence results. It says if you run the subgradient method with a constant step size, then the following will occur: that the best – so the limit is [inaudible] infinity of FK best. By the way, this is the same as lim int of FK, of FK, right? That's what – it's of F of XK. It's the lim int of that, because the best is the [inaudible]. Did I say that right? I did, yeah.

Okay, so this is – so anyway, this goes to F bar. I mean, this thing goes down, because your best function value to date, that goes, that [inaudible] decreasing. So it has a limit, and we're gonna call that F bar. This could be, of course, F star. But here's what you're guaranteed. You're guaranteed that F bar minus F star is less than G squared alpha over two, and that says that if you use a constant step size, then you will end up being suboptimal, but with some number that you can actually, you know, say what it is. So you'll converge to suboptimality, some suboptimality.

Okay – oh, I should say this. If the function is differentiable, then with constant step size you actually converge if alpha's small enough. But for non-differentiable, that's absolutely false, you will not. And it's actually easy to see that. Actually, a good example is just to apply the subgradient method to minimize the absolute value of X, okay?

So let's just do that in our heads, right? So you take absolute value of X. How do you get a subgradient? You're at a point X, and you get a subgradient for absolute value X. What's the subgradient? If X is positive, what's the subgradient?

Student: One [inaudible].

Instructor (Stephen Boyd):One, it's plus one, there's no choice. There's no other answer. If X is negative, what's the subgradient? Minus one. If X is zero, you can return any number between minus one and one. Okay? So for some reason, our code, it's gonna return minus 1.75.

Student:[Inaudible]

Instructor (Stephen Boyd):Hm?

Student: That's what – if you [inaudible] minus point five there.

Instructor (Stephen Boyd):Minus point five. No, you mean zero.

Student:[Inaudible]

Instructor (Stephen Boyd):[Inaudible] If you return zero, if you're at zero, you're making things easy because then the algorithm just terminates. Because if return zero is the subgradient, that's the condition for optimality, and it can quit. No, I don't wanna do that. I'm gonna return minus .75, just because – some sick reason, because who knows? Yeah?

Student: That goes behind what I was saying, that you could choose [inaudible].

Instructor (Stephen Boyd):Yeah, I know what you were saying. Yeah, I'm telling you, it's not – it turns out – everyone assumes this, so you can – well, we can, you can do a project on it. Everyone assumes, they think that this method, you know, it's so – that by clever choice of the subgradient you can do better. That is probably a true statement, but it gets to be a huge, long – I mean, this is on a [inaudible] somewhere that trades off simplicity of code, which is to say one line, with simplicity of proof, which we will get to maybe in, like, two lines, and the actual how well it works, which is slowly.

You can then have methods that are super complicated, try to estimate the best subgradient and blah, blah, blah, and bundle – they get all fancy and all that. Sure, they work a bit better, but they're, like, way longer and all that kind of stuff, so. But still, my basic comment on that is you'd be shocked how little you gain by being smart and returning a so-called good subgradient.

So, okay. So anyway, now let's imagine and make it simple that you return plus one if you're positive, negative one if you're positive, and if you're at zero, you return plus one. And it's a valid subgradient, can't say it's not. Okay? So now you imagine where you are. You're somewhere, and we do a fixed step – let's do a fixed, you know, a constant step.

They're the same, I guess, right? Because the gradient – okay. So then there's zero to point X, and so you step alpha. If you're positive, you go down alpha. Everybody visualizing the subgradient method? So you go down. When you go over, when you switch sine, now you go up alpha, right? And now you simply go back, you just oscillate like this. Everybody got this? Okay, you are not optimal, right?

Because you go there and it says the subgradient's plus one, it says go that way. You step that way, your sine changes. You go the other way. So that's – so you can see the – and that's, by the way, no matter how small alpha is, you will not converge, because you'll end up in an oscillation like this.

Okay. If you have constant step length, you again converge to something that's suboptimal. And here's the weird part. Diminishing step size rule? It converges. So actually, what we won't do is I think today we won't do the – go over the proof of that, which is, like, shockingly short, but I just wanna point something out.

Ready for – here's some diminishing step sizes. How about one over K, okay? So ready for the algorithm? Just put one over K there, okay? And you now have – and now here's the – ready for the theorem? It's this: it converges to the global [inaudible]. That's how stupid it is.

I mean, you have to understand how dumb this is, right? It's basically, you know, for I equals one to whatever, little K equals whatever – little K equals one [inaudible] capital K, it's basically X minus equals subgrad, subgrad F of evaluated at XK divided by K. That's it. That's one line, okay? And it converges.

I don't know if you're getting the thing. I mean, it's fine. You know, you might ask, well, why didn’t we do this in 364? Well, this is kind of cool, but it's – I mean, these things, as you'll see, they're gonna be very slow. They will have some redeeming features, so. But they're gonna be quite, quite slow.

But still, this is very shocking, if you think about it. You also could say – I mean, if you also believe this meta-statement I made last time, which went something like this: that if you can evaluate a function, you can evaluate a subgradient. If you believe that, then basically what this says is you can solve any convex function. Any convex problem for which basically if you can evaluate the objective and constraint functions – we haven’t gotten to the constraint case, so; but we will.

But it says if you can evaluate the objective function, you can minimize it. Period. So everybody – and the algorithm is, like, it's barely a line. It's not a line, it's 15 characters, right? That's it. I mean, it's that stupid. It's weird. So when you first see this, it's kind of shocking. We'll continue this Tuesday.

Actually, now that people are here, maybe I'll just repeat a couple of the things when people were in between this place. So the most obvious new development in the class is that we're here, and televised. But we'll try to keep it informal too as well, so we'll just pretend it's not here.

And if I say anything really way totally off base, we'll just go in to – I'll go to my friends at SCPD and we'll cut it out before it gets posted, and the people at MIT look at it and then I'm in big trouble. Or the police get it, or you know, whatever.

So we'll do that.

We posted homework one, and I don't know that there are any other – maybe there aren't any other announcements, so. Okay, so we'll quit here.

[End of Audio]

Duration: 74 minutes

ConvexOptimizationII-Lecture03

Instructor (Stephen Boyd):I think we're on. You don't – do you have any – you can turn off all amplification in here. I don't know if you have any on. Let me start with some announcements. The first is this, for those of you who are currently asleep and viewing this at your leisure later, we know who you are, so let me repeat that attendance is a requirement of this class, so now that it's on – when we put it on SCPD it doesn't mean you can sleep late. Of course that doesn't apply to our people here, so – okay, all right. A couple of real announcements, actually that was a real announcement. Oh, the first one is we've assigned Homework 2. We have no way of getting in touch with you because for some reason the Registrar has not populated the course list yet, or maybe that's not true today, but it was true yesterday. So Homework 2 we've assigned and these are gonna be pipeline, so we'll produce the homework whenever we feel like it. It will be due roughly a week later or something like that. No matter when we'll – they'll overlap and all that sort of stuff, so you'll be fetching homework three, sort of while you're still processing Homework 1, or two, or something like that. I don't know. We'll see if we can get three deep. We might, actually. We'll see if we can do that. So Homework 2 is actually assigned. Let's see, a couple of other announcements, oh, two are important. One is that sometime this week, maybe even today, we're gonna schedule what is gonna take – play the role of a section. There will be no section of where you go and the TA's review stuff and you ask about Homework 2, Problem 1. We won't even have that. What we will have in the section is once it's scheduled, and we're hoping it, I mean it could be scheduled, I mean as early as this afternoon or who knows what. So please pay attention to email, not that we have an email address to email with you, but assuming we did. The second, we're gonna actually, just the first couple of sections are gonna do nothing but cover, sorta, discussion and projects. So I'll lead a couple of those if I'm around and can do it, and so that's – but you'll hear more about that on the website or via email if it ever gets set up. Now the other one is quite strange. In 364A you might recall we made, I believe we made SCPD history by taping ahead the first lecture, not only the quarter before, but the year before.

So I think we're gonna do it again. We're actually gonna – we will schedule for this class what could well be the world's very first tape behind. So we're actually gonna go back and redo lecture one. Really, I'm not kidding. So that it can go on the web or something like that. Yes it's strange, I know. So at some point, I will appeal, maybe the right word is beg, grovel, I'll do something to get some of you, statistically, to show up to the, what I believe will be, the worlds first tape behind, where we'll go back and retape Lecture 1. So – of course I can't say all the controversial things I said then, but that's okay. That's the disadvantage. Okay, if there's no questions about last time, then I think we'll just jump in and start in on subgradient methods. So far subgradient methods, we look at the – I mean, subgradient method is embarrassingly simple, right, it's – you make a step in the negative in anegative, I'll call the negative, but the correct English would be an anegative subgradient direction. The step size is not chosen, for example, by the methods you would use in a gradient method. In a gradient method, you're looking to minimize the function, then you might use a greedy method to minimize f, or just get enough to sense in f, and that would be via a line, you know, some kind of a back tracking line search, or line search. Step size in subgradient method, totally different, and actually, frankly bizarre. Here's what they are. They're often fixed ahead of time, that's the strangest part, so very often, it's something like this. It's just a constant, or it's a constant step length, or it's some fixed sequence, like one over k, and we'll see, in fact, that there's not really a whole lot more you can do with this gradient method, than this.

That, actually, being smart in choosing your step sizes, generally speaking, I haven't found it to work that well, so okay. So the assumptions are standard. By the way, if you look in various books, you will find proofs where the assumptions are much weaker. I make as much as, as many assumptions I can to make the proofs fit on one page because I think that that's – if you can get that, then you can easily go back and read the four page version in the Russian book, written in English, though, so, and do that. So for example, in particular, you don't need this constraint here, this assumption that that is a global Lipschitz constant on F. That's not needed. That complicates the proof if you don't have it, but that's not our, I guess that's not our point. Okay, so let's jump in. Here are the conversion results. I think we looked at this last time, and then let's look at how the proof goes, so here's how it goes. It works like this. What is gonna go down is the Euclidean distance to the optimal set, not the function value. So if you remember the proof of the gradient method, well there's several things you can use as a Liapina function in the gradient method. You could use the function value itself, so the function value goes down, and then, of course you have to go – once you know something goes down, it's got to converge. The question then is whether it converges to the optimum, but that's a simpler thing to argue. The other option for a Liapina function, or merit function, or whatever you want to call it, whatever the function is that certifies progress. The other option in a gradient or Newton-like method would be the norm of the gradient. That's another possible Liapina method.

Here, what is going to serve as the Liapina function, or the merit function or whatever, is actually the distance to the optum – I'm – what was that? Oh, maybe that was this thing timing out. Okay. So I'll use the manual time out prevention procedure. What's that? Yeah, that's all right, I'll just fix it later. Okay, so the distance to optimum is what's gonna go down. Now actually, there's something quite interesting about this already, and I'll say this now because it's gonna come up as we look at what happens with subgradient method. Let me tell you what's weird about it. If you're running a gradient method or a Newton method or something like that, you know the function value, so you know if you're making progress because you're last function value was minus ten, at your next step it's minus 10.8 and you know you just made, whatever, 0.8 progress because the function value went down by 0.8. Here's the weird part, you don't know this, so although it is gonna be what's gonna be going down, you won't know it. So that's gonna be what's a bit odd about it. Okay, so let's look at this. The argument goes like this, xk plus one is in fact precisely this, that is xk plus one, and now I regroup xk minus x* and I pull it, and I expand this so I get the norm squared to the first term, I get the norm squared of the second term here, and I get twice the inner product and that's here. Now this, in equality, holds, in fact, by the – in fact that's literally what it means to be a subgradient. That's this in equality. Okay? So if you apply this recursively, this basically says that your distance at the next step to x squared is less than or equal to the old distance to the optimal, and then let me point out which terms are good and which are bad. This term is bad because this is an increase in distance to optimum. This term is good because this thing is bigger than that, well by definition of f*, so therefore, that's positive, that's minus two alpha k, alpha is positive. So this is the good part, and that's the bad part. Okay?

So what this says is, unless this term overwhelms that term, you actually don't make progress on each step, but notice what it does say. It does say that when alpha gets small enough, you absolutely make progress towards x*, which you don't know, by the way. So that's what this says. So what happens is, you simply say, "Well look, if this is less than that, I simply apply recursively this inequality. I plug that here and I end up summing these two terms, and what I'm gonna get is this. I'll get this if. If I apply this recursively, I get that the next state minus x*, that's an optimum point, is less that or equal to the initial deviation from x* and that's gonna be less than r squared." That's sort of our assumption, here. Minus twice this sum, here, that sum of – these are positive numbers, plus, and then there's this term over here, and so of course, this is not a good term. That's the good term because it's – well actually it's good, everything works out quite well here. And then I'll make some, I'll do some things here. Let's see. This hasn't changed, but here, I'm just gonna replace these with their maximum value, which is g, okay? Now by the way, if you have a continuous – if you have a function that's differentiable, as you approach optimum, what happens to the gradient? It goes to zero. So, I mean, that has to happen. In fact you can get derive also to balance on that. That's – maybe I should fix that. Or I'll – okay. What's that? Oh yeah, oh yeah. You're right. That's actually what's happening. Hang on. We'll just power this guy up. It's panicking, or it's not panicking, it's just, it's on the battery power save mode, so it's – thank you. That'll – because I thought I changed the thing, okay. Let's see if it stays happy now.

Okay, all right, so for a function that is differentiable, the gradient goes to zero. By the way, is that true for subgradient method? If you minimize absolute value, what do the subgradients look like? Just minimize absolute value x, what are the subgradients of absolute value x? They're mostly plus or minus one. Right? So it is false that the subgradients go to zero. Right? They can remain big all the way to the end, and the subgradient method will be oscillating between a small positive and small negative number, and subgradients will always have magnitude one. So that's the case, that's true for any minimax problem. The subgradients are not gonna go to zero. They're gonna stay large. Okay. Okay, so we get this. Now here we'll use a very crude bound that says that the sum, if I have a weighted sum, these are positive numbers, these are, I guess, these are also nonnegative. Well, I guess if there's, well no, these are nonnegative. One of them is actually equal – no sorry, these are nonnegative. I'll simply replace this with the min of this, and the sum. Now the min of this thing is fk best, by definition, so you get that. And if I put this on the other side and divide by various things, I get this basic inequality. So everything's gonna follow from this one inequality here, but you already see we're in very good shape because if you look at this inequality, here's what happens. You have, this is the best thing you've found so far. By the way, this you know, for sure, right because that you're keeping track of, minus f* and that's less than r squared, that's your original ignorance in distance, plus g squared times the sum of the alphas divided by this, and you can see already you're in very, very good shape.

Then there's a couple of ways we can do to make this work, but now you can see immediately why it would be that the sum and the alphas i's are gonna have to go to zero, in these things because that's the first – some of the alpha i's, sorry, are gonna have to be, go to infinity, so that's gonna make this thing diverge. Okay, so if you have a constant sep size, that's just a fixed alpha, you plug that it and you get this, and if you take a look at that you'll see immediately that as k goes to infinity, you end up with g squared alpha squared over two alpha, which is the same as g squared alpha over two. So that proves that as – if you just have a constant fix size, well, people call it constant sep size, it's not really the size. If you make a constant alpha, what happens is this, you will converge to within g squared alpha over two, as k goes to infinity, so you don't converge, but you will converge to within some boundable distance from optimum. That's constant sep size. Now constant step length says you simply, and by the way this is the classical subgradient method from, you know, this is what you learn in 1971 in Moscow, is this one, right here, or with a, actually it's this with a decaying terms here, where these are gamma k's. But that's the one you learn, that's the one, by they way, that does not need the Lipschitz constant g assumption, but not that it matters, but okay. In this case you get this, you get the best f – f* is less than or equal to r squared plus, and then you get the sum of these things over here, and what we'd do here, we went back two inequalities, you write it this way, r squared plus gamma squared k, and what happens now when you let k go to infinity is that this converges to within G gamma over two, instead of g squared. So you get a slightly better bound or something like that.

I'll say a bit about how the subgradient method works in practice, and in fact, you'll see a fair amount in the next week about how it works in practice. In fact, I guess, you'll also code up some, and try this yourself. Okay, so if we start with the most canonical one is that you have square summable but not semmable coefficients or step sizes, and the canonical one is something like one over k. That's your classical sequence here. That diverges, it's not semmable, but it's square summable, in this case it's extremely simple because as k goes to infinity, this converges to a number. So that just converges to a number, which is the sum to infinity, and this converges to infinity, so obviously that goes to zero. Okay? So that's, so there you go. By they way, this is – since we now actually, you've seen a whole proof, I wanna emphasize how ridiculous this whole thing is. You have just seen an algorithm that is one line at best. You could make it two or three if you added some comments. Okay? It's basically, it calls a get subgradient method on a function, on a convex function, and it gets a subgradient, and it takes, it's basically x, well in fact it's, the algorithm is this, it's embarrassing, it's x minus equals sub get sub grad, f dot get sub grad divided by k, or something like that. That's it. I mean, it's a ridiculously simple algorithm. There's no line search, and it basically computes, it finds the minimum of any convex function, differentiable or not. Okay, so it's really, it's worthwhile sitting to think about it for a minute, how ridiculous the whole thing is. So that's what it is. The proof, as you saw, stretches onto two slides, but that's only, that's expanding it out. It's really, we're talking a paragraph here, with lots of discussion in there about the proof.

So in other words, it's really quite trivial. Now, we're gonna find out, it's got some good properties and bad – I mean as, I'm sure you've already guessed by now, it's gonna be way slow. But that's okay. It's gonna have some other virtues, but simply the idea that there should be universal algorithm for convex optimization that's one line; the proof is barely one paragraph. I still find quite implausible, I mean, the whole thing seems ridiculous, and yet, it's quite true, so. Okay, stopping right here. Okay, now if you wait until this thing, remember this is a bound on fk best minus f*, so now let's leave alone the fact that you really aren't gonna know r, probably. Of course, you could put an upper bound on r there, that's fine. Well, that's not true. You might have some very crude bound on r. Sometimes you can estimate a g, but often not. Right? So if this is – this stopping criterion is a little bit fishy. In any case, waiting for this thing to be less than epsilon is really, really, really slow. It's also often not – I mean, it's just very, very slow. Now you can ask all sorts of questions, by the way, you should be reading the notes on this because there is a fair amount more in the notes than in the slides, where things are discussed in great detail. So you should be reading those. You can ask all sorts of cool questions, like you could take this bound here and you could ask yourself, "What are the choice of alpha i that minimize this." Now, think about what that would mean. It would mean, what are the optimal step sizes. You fix a number of steps and you want to minimize the upper bound on fk best minus f*. Right? That's what you would do if you minimize this.

Now, let's actually talk about that briefly. So we want to minimize this function here over choice of alpha, those k alphas, right? So let's take a look at that. First of all, what kind of function of alpha is this, provided they're all positive? It's what?

Student: Quasiconvex?

Instructor (Stephen Boyd):Oh, I agree it's quasiconvex, but I think it's more. Hint, it's a word that appears in the title of this course. What kind of function is this? It's convex, obviously, right? Now actually, why is it convex, actually? Somebody?

Student: It's quadratic [inaudible].

Instructor (Stephen Boyd):Hey, it's quadratic over linear. It's a quadratic over linear function, and the linear part is positive, so we're totally cool. It's a convex function. Okay, so actually that's very interesting. It means that choosing the optimal sequence of step lengths in the subgradient method to minimize the upper bound on fk best minus f* is itself a convex problem. Now, the question is what's the solution? And I'll just mention this; it has to do with symmetry. If this function here is symmetric, in other words, if I apply any permutation to the alphas, it does not change the value of this function. Everybody agree with that? So that's clear because you just sum alpha squared with sum alpha i, that's all. So it is symmetric. Now that means that if you for – now, what I'm gonna argue now is that the alphas are constant. The optimal alpha is constant. Okay? And this is a general fact about the optimum of a symmetric convex function, can be assumed to be constant. Constant means constant under the symmetry group, okay? So let's see, the argument would go something like this. Suppose you come up with some set of alphas that's optimal. But they're weird, I mean, they start, they grow, they shrink, they grow, it doesn't matter what they do, they're non constant. Okay, and they're optimal for this, okay? Well then, if I apply a permutation, that's gotta be optimal, too because it's got that same function value, which you allege to be optimal. So if I take your alphas, which can vary, and I permute it any way I like, I get something that's optimal because it has the same function value.

Okay, now do this. Take all permutations of the alphas, add them up, and divide by n factorial. Sorry, k factorial. Okay? So you average over the entire symmetry group, the orbit over the whole symmetry group. Okay, but you know what you're gonna get? It's gotta be optimal, and that's by Jenson's inequality because I'm taking an average of things, this function evaluated on the average has to be less than or equal to the average of the function, but the function is optimal every time, so therefore it's optimal, too. But when you average over all permutations, you get a constant. Okay? So that's my proof, complete proof, that in fact the optimal alpha here sequence vector. Vector of alphas is constant. Once you know it's constant, this is really stupid, you just remove the i here and put k in here, and you optimize and you get this, so it's constant, and you get r over g divided by square root k, that's your optimum step size. By the way, you will see this come up in various things. It comes up in the lore, also of – so it's ingrained in the history of the subgradient method. You'll see square root k's all over the place. By the way, is that square summable? Is the square summable? No, of course not because see, squared you get one over k's and those grow like log, so it's not square summable. On the other hand, this is what you would do to get the best bound after k steps. Okay, let's see. Okay, so number of steps required if I plug this into here, is this, and it's actually a quite beautiful number, rg over epsilon I mean, I think we can interpret it perfectly. If you go back to the pert, to the initial point, so the initial point before you've optimized anything, all you have is prior data.

So here's what you have. You have an initial point x1 and you don't know anything, you haven't even evaluated one subgradient. The only thing you know is this, someone has told you that you are a distance, you are no more than a distance R from optimum. That's what you've been told. You've also been told that the Lipschitz constant on your function is G. Based on that information, you can bound how suboptimal you are right now. Because the farthest you can be, the optimal point could be away from you is r, and the Lipschitz constant is g, so rg, this is actually – that is basically your original, your initial ignorance in optimal value. Everyone, that's how – if you were not allowed to do anything, this is an upper bound, probably a terrible one, but nevertheless an upper bound, on f of x1 – f*. Everybody cool on that? So this is your initial ignorance in f*. Epsilon, of course, is your final ignorance in x*. So this is your ratio, this here, is the factor by which your ignorance improves over the algorithm. It's literally, I mean if f is in dollars or euros or whatever, something like that, it's a cost. This is literally the ratio of your, actually this is your prior to your posterior ignorance, and it says that the complexity is gonna go like the square of that. Okay? So that's the – everything's interpretable here very, very nicely, so. Okay.

Now, I'll tell you the truth. It is cool because the proofs are so short, to look at the proofs and things like that, and actually good things have come from the proofs and things like that, like an understanding of what these step lengths mean and all that sort of stuff. But the real truth is this, when you actually use a subgradient method, and we will see applications, actually we'll get into stuff later today, where you'll start seeing applications of it, and then over the next several weeks, you'll see serious applications of it. It's gonna come up in areas like distributed optimization. It – look, it's so slow, it has to have its positive qualities or we wouldn't be here. So anyway, we'll see what they are. Okay, so – but here's the truth. The truth is there really isn't a good stopping criterion for the subgradient method. That's the truth. Unless there's something else in your problem, I think at one point, I don't know, someone over that way asked about cleverness in finding a subgradient, if you have just a separate method to calculate a small subgradient, or whatever, you might get lucky, or something like that. But the truth is, in general, there's no good way to know when to stop in the subgradient method. And this bound, all the theoretical bounds are like, way – not too useful, and so on. Okay, so let's look at an example. So here's an example. It's piecewise linear minimization. So you wanna minimize this function. Now, I should add, here's how we would do this. I mean, everyone here, you would to it – I mean, this you would do by, obviously you would do this by, this would be very simple, you would do this with some – you would convert it to an lp and solve it. And nothing could beat that. Absolutely nothing could beat the efficiency of that, no matter which method you choose to solve the lp wouldn't make any difference. That absolutely couldn't be beat.

So it's to be understood here that this is not a recommendation or endorsement of this method for piecewise linear minimization. So okay, so to find the subgradient for this max is very simple. You evaluate all these functions, you gather the function values, and then you find out, you pick any one of them which is at the maximum. Okay? So you just pick one that's at the maximum. If there's only one, no problem, you just pick that one, and we'll pick that j, and we'll take g as aj because it's the gradient of the one that's height at the optimum. And the subgradient method looks like that. So again, it's just embarrassingly short method, and let's see how that works. So here's the problem instance with maybe, what 20 variables, it's a piecewise linear function with 100 terms, so it's the max of 100 affine functions in r20, and it turns out for that instance f* is about 1.1. I should add that all of the code for all of this, I believe almost all of it, I think, maybe all of it, is on the website. So on the website there's the lecture slides, there's also a tiny little link for notes, you should read those. Those are the more detailed notes, and then after that, it should be all the source code for everything, so if you wanna see what we did. Okay, so here's affine, it's f*. Not an impressive scale. That's not ten to the zero and ten to the minus eight, as it would be if we were looking at an interior point method or something like that. That's ten to the zero, and that's ten to the minus one. So the bottom of this thing is basically one, 10 percent optimality. Okay, so okay. So this is constant sep size with various step sizes like this, and here it is on a bigger scale. Note the number of iterations here, and it looks – it's doing something according to theory. This is something like, that's 10 percent and that's 1 percent. And actually some of these are not at all bad.

For example, this method right here, and you know, we do have to be a bit fair here because in fact, all we're doing, each step that computational complexity, is we have to evaluate all those affine functions. That's actually order Nm, right? Because you multiply an Nm matrix by a vector, that's all you have to do, is m by n. So the cost – each iteration here is order mn. There's then, of course, there's a log m step, where you calculate the maximum, but that doesn't count, and then there's a little order n step. Mn is what you multiply with here. Now, if you do an interior point method, what's the complexity of this? If you use an lp to solve this problem, what's the complexity? Total, actually. What is it? You wanna hear the pneumonic for this? The pneumonic is this, if you know what you're doing, it's always the big dimension times the small dimension squared. That's if you know what you're doing. If you don't, it's the big dimension squared times the small dimension. Okay? So you wanna do least-squared – because basically look, each step in this, you have to form something like, let's see, a is like that, so you're gonna form a transposed da, and that's what you're gonna have. You're gonna have to form that hash in and then solve it. Actually forming the hash in is gonna be the one that's gonna cost you the most, and that's gonna be order mn squared, here. So compared to order mn, it means we're off by about 20. Now this is very rough. It's disturbing to me that you're looking at me like that, so. I knew went fast in 364a, but this is the time for you to go back and read it and figure these things out. So basically, and that's what comes from looking at these things, too. You all coded this. So don't look at me. Like, you've done this. So anyway, it really is, trust me, it's a least-squares or a normal equation solved. In fact, forming a transposed da is the dominant, is actually the dominant computational effort, and that's – if you know what you're doing it's mn squared, so.

So you're off by about n, and that means that in these things here, you really should divide by twenty, minimum, to get – and then these things aren't so bad. That's – well they're not so good, either. So you divide by 20, what do you get. 60? Okay, so on the other hand, an interior point method by this effort, and we're being very rough here, by this amount of effort, 60 steps, it's way over. I mean, you've got accuracy to whatever, ten to the minus 12 at that. You're done. You're double precision stationary after 60 interior point steps, right? So the point is, these are slow. What's interesting is, this one actually is a little bit attractive. It's almost attractive if what you wanted was something that's like, 5 percent accuracy or something like that. But okay, this is – so anyway. Bottom line is that shouldn't – it's not fair, I mean you have to, don't compare this to the 40 – the 30 over here you'd see for an interior point method, but this is more like 60 or something like that, so. Okay. This is what it looks like for various other rules. The first one is 0.1 over square root k, and the other one is the square summable rule, just one over k, and that gives you these. Now actually, there is something to note about this, and in fact it's probably a very good thing to know about subgradient method, and this holds for gradient methods, as well. You get, for a little while, you do well. In other words, you do pretty well after a couple of steps. You get some serious reduction, and then it just jams. Now theory says that this is gonna go to min – this is gonna just go to zero, right, so. But it's gonna go really slowly.

In fact, if you read the notes, you'll see that these methods can't go any faster than one over square root k. That's very slow. So if you're looking for high accuracy and you can avoid a subgradient method, that would be a really, really good idea. So, okay. But this is not bad, right. That's 1 percent – that's 10 percent, that's a couple of percent here, and in fact for a whole lot of applications, a couple of percent is just fine. Okay, now we're gonna look at what happens if you know the optimal f*. Now, when you first look at this you think, sorry, that's a bit circular. When you minimize it, how on earth would you know what the optimal value is? What we'll see, actually, it seems ridiculous, but there's plenty of applications in cases, you'll see them in a minute, where you actually know the optimal – you know, often the optimal value is like, zero, basically, and you know it's zero, and what you just wanna do is find a point where f of x is zero or is near zero. Okay, so. But when you first hear this, it seems quite absurd.

And the optimal choice, and this is something to do to Polyak, although I talked to him about it, and he said, "Oh, no, no. Everybody knows that." So who knows who it's due to. He would tell us. Okay, so. And the motivation behind this step length is this, and I'll draw a picture to show what it really means. The motivation is this. This is the inequality we used at each step and it simply – let's do something greedy and take the right-hand side and minimize it over alpha k. If you minimize this right-hand side, you get that. Now of course, in many cases that's totally useless because of course you don't know what f* is, but if you happen to know what f* is, minimizing the right-hand side here is, I mean, it's a reasonable task. Let me show you what it's based on. It's based on the following idea, if you wanted to get a picture for what this would be. It basically says the following, you're minimizing a function now – I've drawn it differentiable, and you're here. I'll draw it differentiable, but it – that's okay. And you know f*, don't ask me why, but for some reason you know f*, and this step length basically calculates the point here. It calculates that point there. So what it does is it simply extrapolates the function affinely, right, I guess most people would say linearly, finds out when it hits this optimal value, which has been conveniently supplied to you, and then that takes a steps length that does that. Okay, so that's the picture. Oh, let me ask you a question. If a function is piecewise linear, how well – let's talk about how good this – let's slow down. This step size, now that you have the picture in your head, how does it work for quadratics? How does it work for quad – I just drew it. So how does it work? How does it work for – just do it in your head. Just take the quadratic, take a point, draw the tangent, okay. So how does it work? How fast does it converge for quadratic? What? Well, you make about half – I mean, I'm just talking about x squared. I'm talking about how do you minimize x squared, using the Polyak point of step length? The answer is, you kinda make about half – you – x gets divided by two each step, right, roughly. So you make – your converging, it's pretty good.

Oh, by the way, you can easily figure out ways to make this method work very poorly. You need a function that's very steep. So if a function is very steep, this is gonna be really – and then, is really steep, and then has a kink, and then goes down slowly, until you hit that slow part, its gonna take you zillions of steps. Everybody see that? Because you're just going down, you're going very slow. Okay, how about piecewise linear? How about – is this a good idea for piecewise linear? How does this work for absolute value, by the way? One step. How about piecewise linear in multiple dimensions? What do you think happens?

Student:[Inaudible].

Instructor (Stephen Boyd):Yeah, okay. Yeah, it depends on the number of pieces. But actually what happens to piecewise linear, this algorithm terminates finitely. Take the finite number of steps because once you get – if you go in on a piecewise linear function and you look way down at the bottom, where everything comes together, at that point, once you're on one of those faces, next step it's all over. Right. So this method for piecewise linear actually works well. So this is a step length rule that's kind of appropriate for nondifferentiable functions. It's actually not that appropriate for differentiable ones. Okay. Now when you plug in this optimal step size, here's what you get – oh, when you plug in the optimal step size, what's great is that, now, the distance for optimal actually goes down every step, so this negative. In the subgradient method it need not, which all you can say is that eventually it goes down, and the fact is you don't even know when it starts going down. You don't know x*, you don't know that, and so on. Okay? Okay, so in this case, your distance actually goes down every step and by an amount – I mean, it basically, it's fantastic, and you can see immediately everything's gonna work out in your favor. What happens is this; this thing goes down by an amount that depends on how suboptimal you are. Therefore, if you're suboptimal, if you're quite suboptimal, the distance goes down a lot. I mean, this is kinda just what you want. If it doesn't go down much, that's because you're nearly optimal, so it's perfect.

Okay, now if you just apply this recursively, you get this, and what you get here is this, is I can replace this with g, and then put g squared over there, and I get that. So this is the sum of the sub optimality squared, is less than r squared g squared, by the way, that's a faster convergence than the one over k, one over square root k, or whatever. It basically says the convergence is l two, right, that the sub optimality goes down and is square summable, right, so for example – well, anyway, it can't be as slow as one over square root k or something like that. Now of course, that's because you know f*, here. So that's how this works. So here's the piecewise, same piecewise linear example, now of course, in the piecewise linear example, there'd be no reason that one would know f*. That's ridiculous, right? So you wouldn't know f*, so this is unfair, this is just to see what would this – what advantage would you get, and I think, this is the Polyak, the so called optimal step size is this thing, and you can see it does as well as the – as any subgradient method is gonna do. And that's what this one is. Again, it's not terrible. That's what it is.

Okay, so now we're gonna apply this to a very famous problem. It's the problem of finding a point in the intersection of convex sets. So very, it goes way back the problem, it's used in lots and lots of fields, and the method we're gonna talk about – well, not gonna talk about, it's gonna pop right out of subgradient method, is very classical, and it's used, actually, to this day in lots of fields. In fact, in lots of fields they haven't grown out of it. Someone discovered it in the '60s and they just never went farther. They just arrested development at this method. So I'll name some of those fields later, maybe. Okay, so here's what we want to do. We have a bunch of convex sets. They intersect, and what I wanna do is actually find a point in c by minimizing the maximum distance to all of them. Okay, now this is a convex function here. It's a convex function, oh and if I minimize – oh, by the way, what's f*? What's the minimum of that? Zero, good. So this is a case where you know f*, and it's zero because this is not empty. So there's an x in all of them, there's an x that satisfy all of these are zero, so f is zero, max of distances don't get smaller than zero, so there's not – f* is zero, okay? And we're gonna minimize f using subgradient method. Now, how do you calculate a subgradient of the max of distances to a bunch of convex sets? Well, the way you do it is this. You evaluate all of these, so you evaluate all of these. You figure out which, you choose one that is maximum. Doesn't have – if there's pie, you pick it, break it arbitrarily, makes no difference. That's the subgradient method. So you break it arbitrarily, pick one that's maximum called j, and then actually, the distance to a convex set, in fact, if it's provided its positive is differentiable, always. Let me think about that. That's true. It's differentiable. Let's see, yes, that's true. If it's Euclidean distances, it's gonna be differentiable.

So – and the gradient is very simple. It's actually a unit vector that points in the – from where you are, well, let's see. Let me see if I can explain it. Here's you, here's the convex set. You calculate the projection of that point on the convex set, and then the gradient points exactly in the opposite direction and has unit length. It's kind of obvious. It says basically to go to, to move away from a convex set at the highest rapid, as fast as possible. You should calculate first the point on the set closest to you. You should turn exactly around and go in the other direction. And if you ask how far, if you go three millimeters that way, how much farther are you from the set? The answer is three millimeters, locally, right? So that's, I mean, you could also do this just by calculus or whatever, but the point is, that's the gradient. So it's a unit vector. Okay, now we're gonna use Polyak's step length, so we just simply update like this, and let's see. Oh, here's what we do, so I – this is the f minus f* and so on like that. And I simply get this. It's a – in fact if you work it out, it's quite – it just works out perfectly. Everything works out. This is norm, it's the maximum distance, and that happens to be if you pick j as the one that was farthest.

It's norm, it's the dist – it's actually the norm, it's this expression here, which is the same as for the, it's the one for j here, but this is what that is, that [inaudible], this goes this minus that, that's gone, and the only thing you're left with is a projection, so it's that. So it's really dumb. So here's the algorithm, ready? You have a bunch of convex sets, and – let's see, so you have a bunch of convex – let me just draw, I'll just draw the most famous pictures for two, but it doesn't matter, I can draw it for, you know, three. Here's this, and then I have this. And the way it works is, you're here, anywhere you are. Actually you could never be there, but that's another story. So let me assume you're here. Let me assume – whatever, you start here. And what you do is you calculate the distance to each set, so that's the distance to the first one, that's the distance to the second, and that's the distance to the third. The largest one is this one. So you simply project the point onto this set. It's that stupid. I mean really, it's that dumb. Then you're here, and now you calculate the distance to the three sets. Well, what's the distance to this set? Zero because you're already there. What's the distance to this set? Zero. This set is positive, and so you'll go here, and actually what happened? Did we do it? Yeah, is that right, yeah, I think. So in two steps, this one worked. This one converged with, in a finite number of steps, you know, two. Okay, so this is the picture. So when – by the way, this is called alternating projections because if there's two sets, it's extremely simple. You have two sets, you just project the one, then the other, then one, then the other, and you keep going back and forth. It always has to be the other one because after you projected onto a set, your distance is zero, so if you wanna know what your maximum distance to the two, it has to be the other guy. So that's alternating projections.

Lots of variations on it, too. So that's it. And our basic result says this, that our distance goes to zeros, k goes to infinity, in fact we even have that sum of squares bound and all that, so. By the way, it can be quite slow, this method, and there's some very interesting acceleration methods for it. But this is it. Very famous algorithm. So here's the picture for two. It goes like this, you start here and you project there, then there, then there, then there, and so on. That's x*. Now in fact, the theorem, it does not say that you get convergence in a finite number of steps. What happens is, you get – what happens is your distance, that maximum goes down. That's what happens, and it goes down to zero. That's guaranteed. So basically it says, if you run it long enough, you're not very far from the farthest set. So that's what it says. There are methods to – there are variations on the alternating projections that will guarantee that you find a point in the intersection, provided the intersection is not empty, and they'd be based on things like this, basically over projecting. You back off each c slightly. You shrink each set slightly, so that they still have intersection that's not empty. Then you apply this algorithm to the shrunk things, and so you – and then, now what happens is, our method says, or the general theory says, subgradient method or alternating projections is gonna – your distance to the shrunk sets goes to zero, but that means after finite number of steps, you're actually in the interior of the other steps. So what I just said was actually correct, I mean, sounded informal, but it was actually correct, so. Okay, so that's alternating projections.

We'll look at an example and then I'll mention some other examples. Let's see, let me just mention some examples. It's used in a lot of cases, so you're gonna actually – so what you're gonna want to do, and this is actually not a bad thing to, everyone should do this anyway, is you need to catalog in your mind the sets that you can easily project onto, with low complexity. So actually, all educated people should know that, anyway. So that's – you just need to know that, not just for algorithms, but for everything else. So let's actually talk about sets you can project onto. How about affine sets? Can you project onto an affine set? Yeah, sure. That's least squares, come on. So you can project onto an affine set, that's easy. How about projecting, now let's see, how about projecting onto a ball, a Euclidean ball? That's extremely easy, right? You, wherever you are, you look at the center and you go towards it and when you hit the surface of the ball, you're there. Okay, let's see. How about some – how about the nonnegative orthant? How do you do it? Yeah, you just truncate all the negative numbers.

Okay. Let's see, I can do a couple of others. What would be some other ones? How about, let me go, let me – I think it's right. How about unit box? Or just a rectangle, box? How do you project onto a rectangle? God I hope this is right. Had a weird feeling. But how do you project onto a rectangle? Yeah.

Student: They match that with –

Instructor (Stephen Boyd):Yeah, if you – you saturate it. Right, so you look at coordinate, you do it all separately, you take coordinate x3, it's gotta a lower, there's an l3 and a u3, and you said x3 is equal to x3, if it's between those two, otherwise, if it's below l3, you say it, that it's x3, if it's above u3, it's u3, so you saturate it, I guess. I hope that's right. Anyway, I think that's right. Okay, there's some other sets you can project onto easily. There's a bunch of them and it's actually good to have on your mind a catalog. By the way, you can project on the lot of sets, depends on how much effort you wanna put into it. How do you project onto a polyhedron? How would you project on a polyhedron? I mean, it's not gonna be an analytic solution, but how do you do it, in general? A QP, so you solve a QP. You solve a quadratic program to project onto a polyhedron. So that's one. And we can talk about – so for example, there's some non obvious projections. We should put that on the homework. So project onto the unit simplex, for example, is a non obvious one. It's – that's not true, it's a homework problem. But anyway, I forget it periodically and then refigure it out, and then swear that I'll remember it, and then I forget it, and so one. Anyway, okay. Alternation projections, okay. We'll look at an example, it's positive definite matrix completion, and let's see how that – what that is. What happens is, I have a symmetric matrix and I have some entries are fixed and I want to find values for the others, so that I form a PSD matrix.

Now, the truth is, if you were in a solving problem, you'd really wanna do something else. For example, you might want to complete the matrix in such a way that the determinate is maximized. That gives you a unique point, in fact that gives you a beautiful point, it gives you a point when you complete a matrix in order to maximize its determinate, that's got all sorts of names, I guess in single processing it's called, like the bird maximum, something or other. In statistics, it's got some name, too. I forget what it's called. It's the maximum entropy completion, or something like this, but you get a matrix whose inverse has zeros in the entries you can adjust. So I promise you that, that's just straight from the optimality conditions. But we're doing something simple. We're just doing feasibility, and so if you like, you could think of it this way, I'm gonna give you some samples of a covariants matrix and you are to fill in the other entries with merely numbers that are plausible. That's – plausible means it's positive semi-definite, so, okay. So here's what we're gonna do. The first projection is gonna be on to the positive semi-definite cone. That's an easy projection. That should do – that's also a good thing to know. As you project onto the positive semi-definite cone, a symmetric matrix, by just truncating the igon value. You just truncate negative igon values. That's your projection. Okay? By the way, there's lots and lots of things like this, right? So for example, how do you project a non symmetric matrix onto the set of, onto the unit ball in spectral norm? That's the unit, the maximum singular value. So your set is the set of all matrices with maximum singular value or norm, less than or equal to one. Guess. I mean, if you know the answer, fine, but just go ahead and make a guess. How would you truncate – how would you do that I gave it away, did you notice that?

Student:[Inaudible].

Instructor (Stephen Boyd):Yeah, you truncate the singular value. So you take the SVD of the matrix, if all the singular values are less than one, you're done. You're in the ball, there's nothing to project. I mean, you don't have to do anything. You are projected. Otherwise, any singular value that's above one, you replace it with one, and you have a new matrix now. You have u sigmatildy v transposed. That's the projection. Okay? So anyway, projection is not, it's a good thing to know, to keep in your mind, what are the projections you can do cheaply, and at what costs, it's a very good thing to know. Okay, now projecting onto the set c2 is completely trivial. That's the set of symmetric matrix, has the specified entries, how do you project onto that? You just simply take all the entries that are fixed, and you reset them to the value they're supposed to have. Okay? So here's an example, it's a 50 by 50 matrix, so about half its entries are missing. So think of this as a covariance matrix, and I give you all of these entries, and you're to fill in the blanks with non – with just entries that are consistent and so on. That's the picture. And by the way, the alternating projections obviously is completely trivial, it calculates the singular Vigee composition. It wipes out all the negative igon values, reconstitutes the matrix, it's not positive semidefinite, but you messed up all the entries that were fixed. Now you go back and you fix all the entries that were fixed, but now you have a matrix that's non-positive semidefinite, and you keep doing this. And the amount you correct each step goes down. Oh, by the way, you'll find this and related methods in statistics and machine learning, you'll find these, things like this all the time, in things like EM algorithm.

You'll also find them when you deal with missing data values, is a very common, you'll see stuff like this where you alternate, you sorta guess some numbers for the missing data, then you analyze, then you go back and replace them, and you know, like that, back in. So the interesting part is some, but I have to tell you most of those problems, some of the problems are convex, most are not. So, okay. Okay, so here you get convergences linear, it's not a big deal, but. By the way, this means that we have not produce a positive semidefinite matrix. What it does mean, is depending on whether you taken an even or odd, if you take an even or odd iterate, the even ones have exactly the right numbers in the fixed positions, but it's gotta a tiny negative igon value. The odd iterates are positive semidefinite, but the entries in the fixed positions are not quite what we ask them to be. Everybody see what I'm saying? That's just because you're alternating back and forth.

So if you wanted to project into it and have finite termination, you could over project in the positive semidefinite cone, and you would do that by truncating igon values that were smaller than some. You'd over project. You wouldn't project onto the positive semidefinite cone, you'd project onto the cone where the igon values are bigger than epsilon, and you could pick your favorite epsilon, like one e minus three. So you'd actually – you would eliminate positive igon values. This would actually then terminate finitely that method. That's your over projection method. Okay, so we say a little bit about speeding up subgradient methods, so the truth is, and I've said this many times and you'll see it when you do this, these are basically really slow. I mean, they're – I have to qualify that. They're really slow, they depend on the data, all sorts of things about them. By the way, that can be wrong – it turns out, if you're lucky for your problem, class, if you're looking for 10 percent or 3 percent accuracy, these methods can work fine. Just absolutely fine, if you're lucky, especially with some kind of acceleration method, I'll talk about them in a second.

You might ask, “Why would anyone use these?” And we’ll see, soon because there are gonna be some methods that work, that these things work with. For example, they generate completely distributed algorithms; we’ll see that in a minute, which is very cool. Okay, but basically what happens is, these things jitter around too much, and let me just draw a picture to show how that works. And there’s very famous methods about this, long discussion of it, and the truth is, I don’t really fully understand it all, so in fact, this would be an outstanding project. Someone just – you have to have the right kind of, slightly weird personality, but if you did, and you just decided, that’s cool, I’ll delve into this, I’ll read all this – you don’t have to read Russian, it’s all been translated, so you could actually work out, I mean because I don’t actually know what best practices are for subgradient method. By the way, it is topic of current, I mean people are working on it now, coming up with new stuff, and it’s really actually quite cool and interesting, so it’s an absolutely current research topic, and I – if someone wanted to work on it, it’d be very cool. Okay, so let me show you the problem. Let’s project onto two sets, this guy and this guy. And by the way, it kinda looks like this no matter what because in the end game, in the intersection, this is the intersection. Did I do that right? Yeah, sure. Okay, let’s say that’s the intersection. Is that right? Yes, that’s the intersection, okay. There’s the intersection, and you’re here. Whether you go like this, you go like that, and then you go like that, you go like this, and you get this thing. Right?

So, and in fact, you can seem something now that all you need to do is take these things and if you make them, kind of, if you make them approach each other, more and more close to tangent, this subgradient method just slows to, like, zip. Anyway, it’s gotta work, but the point is, it slows to something very slow. Now, when you see this, there’s some kinda obvious things. I know this is kinda like, it doesn’t really make any sense, but – what – I mean, this tells you something, here. It looks like a lot of un things, too. It looks like the gradient method, I don’t know if you remember that. But the gradient method does the same thing. It kinda goes in the wrong, it does this. And what’s a suggestion here, for how you might, what you might, what you wanna do? What’s that?

Student: Overshoot, when it’s speed up?

Instructor (Stephen Boyd):You could overshoot. Actually, I would say that the – maybe even – there’s a lot of things you could do. Overshoot would be one, but the main thing, actually, would be that if you smooth the subgradients, if you low pass filtered, or smooth the subgradients, you’d get the right direction. Because what happens is first you go this way, then you go that way, then you go this way, that way. And what happens is, this is just sort of like thermal loys, it’s not really helping anything. You get a very small drift this way, so if you were to just smooth these things, you would do well. There are very fancy methods for this, there’s a method called Dijkstra's algorithm, so it’s Dijkstra's algorithm. Not the Dijkstra of computer science, it’s a different one, so there’s multiple Dijkstra's algorithms, obviously. So there’s a method here, and what it does is it looks at two and then you really wanna kinda look at two of these, and then kinda go along that direction because that’s kind of where you wanna go. So it’s something like, you wanna go in the direction, you wanna look at where you’ve been, like, you wanna go back and forth, one or two, and then go in that direction. By the way, this would work pretty well, too. So the basic idea is you wanna smooth things, and so, and you can see if you – this is one. This is called the heavy ball method, I guess that's direct translation from the Russian, I guess. Well, I know it. That's where it's from. And this says the following, I mean it's really wild, it just says, "Yeah, go in the negative subgradient." But then this says, "Give yourself, also, a little bit of, I guess this is momentum, but in a heavily damped oil." And I think that's the correct model. And it says, "Sorta, keep going the way you were going with a small factor, but then modify it this way."

And I think that will do it here pretty well. It will smooth it down, that's one. Now there's all sorts of other methods. There's – we're gonna look at a whole other class of methods that use the subgradients, but are faster, that some of them are theoretically fast. Another one is this idea of conjugate directions, that's sort of what this is, where you look, where you make a two term update, and these things are fast, as well. Okay, but here's just to be specific about it, here's a couple. Here's one, is filtered subgradient, so this is what anyone in, I guess if you did signal processing or something like that, this is what you would do, is what you do is you'd go into direction s, but s wouldn't be just the last gradient. It would actually be a convex combination of the current gradient and the last direction.

You can see this is actually not much different from the heavy ball method. It basically gives you a two term recurrence, it just smoothes, it gives you some smoothing, like this, and this is one method, that's filtered subgradient, and then we poked around and found something that some Italians came up with, which was basically – again, they all have the same flavor. The current direction you'll go in is a linear combination or whatever, a positive linear combination of the current step size and the last direction. So that, and they give a very specific beta, it should be that, and gamma is some number in this thing, is something in between zero and 1.2 and they recommend 1.5, whatever that means. Yeah, so here's our PWL example, again, and this is what happens if you, here's the Polyak step length. This just – you'd use the current gradient, subgradient. This is the Italian step length here, I guess, and this is just if you put in some smoothing or something, so. So these things get better, but not radically. I mean, it depends. If you're looking for, sorta, 1 percent accuracy in your application, this could actually be very, very nice, so. And you will see soon, there are many actual applications of subgradients, of subgradient methods, so actually, being able to do subgradient methods efficiently would translate immediately to lots of things. For example, it's gonna come up in flow control protocols and all sorts of stuff in networking and stuff like that, so. Okay, is this gonna work? No, that wasn't right. And let's see. And then we'll start the next one. It's constrained. Okay, so we'll now look at subgradient methods for constrained problems, and I'll stop – I'll just get a little bit into this and we'll talk about it. Okay, so far we've just been talking about subgradient method for minimizing unconstrained minimization. By the way, unconstrained minimization, you can already do a fair amount with just that because these things can be nondifferentiable, so you can choose any, I mean, you can do all sorts of stuff. But it turns out that constrained is no big deal in this case. So the most famous method there is the projected subgradient method, and it looks like this. I mean, it's really simple. It goes like this, you do a subgradient update, completely ignoring the constraints, and you project the result back down here.

By the way, if you said in English projected subgradient, you would actually think of a different thing. You would actually think of the projection applying to the gradient, okay. That's not – so the English phrase, projected subgradient method, is – it's not the subgradient that's projected, it's the subgradient update that's projected. Okay? So, by the way, if the projection itself is linear, which happens if you're projecting onto a subspace or affine set, then they coincide, but otherwise it's not. Okay, so, okay. So this is it. So I mean, it's really dumb if you wanna optimize over positive variables, for example. Just optimize, minimize f of x over positive x, the algorithm is embarrassingly stupid. It looks like this. It's basically x minus equals one over k, f dot get subgrad, period, next line is pos project of x. That's it. So then you just remove its negative entries. It's just that simple. That's it. So these are very, very simple methods, and so it just looks like that, but you can do more sophisticated ones, of course, too.

Okay, so you get the same convergence results, this is described in the notes, and the key idea is this, is that your – each of the steps, the subgradient step and the projection, each of them, when you do the projection, you don't increase your distance to the optimal point. So in fact, that's very important to know because whenever you, you know, when you have an algorithm and you have a proof of something and it works, or whatever, basic rule of thumb is once you have a Liapina function that is, or a merit function or a certificate, whatever you want to call it, a potential function, that's another name for it, once you have that, it says you can modify your algorithm in any way. You can do anything you like as long as whatever you do doesn't increase that function. So if you have some uristic method, you can apply it, you can [inaudible] apply it at any step you like, as long as you do not increase that function. Okay? So that's the – that's actually the practical use of having a merit function or a Liapina function or something like that, is that you can now – you can insert stuff and you can insert it with total safety in an algorithm, okay? So in this case, it basically says the following, it says when you do that subgradient step, of course you get closer to x* because everything else is the same, when you project back onto the feasible set, you don't increase, you can't increase your distance. You actually move closer by projecting to every feasible point, therefore you move closer in particular to any optimal point, so that's how that works. I mean, you can actually work this out to see how this works.

Now let's look at some examples. So let's look at linear equality constraints. Here you wanna minimize a function subject to x equals b. Now projecting onto a solution of a linear equation that's an affine set, that's easy to do. That's just z minus, and that's the projection like that, and so you could write it this way. That's the projector, here. And this is p of z. Do I have too many – I'm getting that feeling of unbalanced parentheses here, but is that just from my angle? No. Okay, so it's quite disturbing from down here when you look up at this angle like this. See it's less – oh yeah, no, that's a mistake. Okay. So the projected update, subgradient update, is actually very simple. Oh, by the way, the projector now is actually affine. Okay, so for an affine function, they commute. If you have – notice the projector of alpha minus, or x minus alpha g, you can apply it to the separate ones, ones separately. And if you work out, if I simply apply this projection to this thing, here's what's gonna – it's gonna end up being this. So you only, what you do is this, it’s the same as taking x, taking the gradient, the subgradient, and projecting that onto the null space of a, and then stepping this way.

So this is probably what people think of when they talk about a projected gradient, or projected subgradient method. But it’s only for equality constraints, so. Let me explain that. Now, you’re at a point that satisfies ax equals b, you wanna minimize f, so you need to take a step in some direction, okay. Well, a safe step direction is anything in the null space of a because that’s gonna preserve your ax equals b. So this says, find a subgradient, ignoring that, and you project that subgradient onto the null space of a, that’s thing, and then that’s now a feasible direction. So you might call this, like, a feasible subgradient direction, and that’s, this is the – so this is the very specific one, here. And this is a quick example; let’s minimize the l one norm subject to ax equals b. Here you can work out what happens. It’s quite straightforward. At each step you do a projection, like this. You take the sign because that’s the projection of x, is the sign, here. And you simply, or that’s a subgradient, I should say. If any of the x’s are zero, then you just can take zero if you like. It doesn’t – you can take, actually any, your favorite number between minus one and one because it will make no difference. Any such thing as a subgradient, you project this onto the null space, that’s this, and you do this. Now, by the way, this would be of really not much interest unless you hade a problem where you can calculate this super fast. So for example, if you’re doing medical imaging or something like that, you’ve got ten grad students and a whole lot, and 90 papers and stuff like that, behind some ultra fancy, multiscale, multigrid, you know, PCG type method, to solve some humongous least squares problem because that’s what this is. Then it says, no problem, you’ll just plug this in, and that’s how you’ll do it. This would be a bad choice, by the way, for something like that, but it doesn’t matter. And here’s how it works, and once again we see that it is slow. So that’s – surprise, surprise, right. So that’s okay, I mean, come on. What do you expect for an algorithm that is one line and whose proof is two lines? Okay, three. I mean, come on. Something’s gotta give. Okay, we’ll look at one more thing – or maybe, actually, maybe what I’ll do is I’ll quit here. This is actually, it’s gonna start getting very interesting so we’ll start there Thursday.

But actually, I wanna make a few announcements because the people who came in after we started. Two important ones, we’re actually gonna – hopefully the Registrar’s email list will be up and running, otherwise we’ll have to do it through access, but please watch your email because we’re gonna announce both the section and believe it or not, a tape behind session. We’re actually gonna go ahead and retape Lecture 1. Please come. You’ll get extra credit or something like that, whatever that is. And this section is gonna be just discussion of projects. Just open, totally disorganized discussion of project, but it should be fun for everybody, and useful. Okay, so we’ll quit here.

[End of Audio]

Duration: 75 minutes

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

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

ConvexOptimizationII-Lecture06

Instructor (Stephen Boyd):All right. We'll start. The first thing I should say is that we're about to assign Homework 4. How was Homework 2? That was due Tuesday, right? And you know we have a Homework 3 assigned. We're gonna assign Homework 4 so that we're fully pipelined. We have at least two active homeworks, typically, at once. What's wrong with that? That's how you learn. You're learning, right? Sure you are. And so the other thing coming up soon is this business of the project. And on the website, we have some date we made up before the class started. And I believe it's even tomorrow. It's tomorrow. So by tomorrow, you should produce some kind of two- or three-age proposal to us. And we'll iterate it. If we don't like it, we won't read past the first paragraph. We'll just give it back to you and you'll keep working on it until we like it. So the way we'll like it is it's got to be simple. There's got to be an opening paragraph. In fact, we were even thinking of defining a formal XML scheme or whatever for it. But it's got to be a paragraph that has nothing but English and says kinda what the context of the problem is. So for example, I'm doing air traffic control or I'm blending covariant matrices from disparate sources or something like that. And says a little bit about what the challenges are and what you'll consider. Then when you get into what the problem is, you describe kind of what it is and what you're gonna do. If it goes on for more than a page, if the setup is more than a page, it's gonna be too complicated for the project. Your real interest might be complicated, but then that's not gonna be a good project here. So what we want is we'll read these and we'll get feedback to you pretty quickly, we hope, on these things. So you should also be just talking to us, just to kinda refine the ideas and stuff like that. And things we'll look for is, it goes without saying, this will be done in LaTeX. Nothing else is even close to acceptable. I won't even name alternatives.

The other is that the style should look something like the style of the notes that we post online. That should be the style. There are lots of styles. That's one. It's a perfectly good style. So I don't see any reason why you shouldn't just attempt to match that style. That's statistically for random style checker. No statistical test should reveal that, for example, I didn't write it. That would be a perfectly good model. There are other styles that are very good and different, but that's a good style and you could go with that. So if you look at your thing and if you find that your notation for an optimization problem differs from mine, change yours. Because, although there are other consistent notations, like there's the Soviet one, and there's other ones, I don't need to see them. It's just easier. And if used consistently, that's perfectly okay. But just to make it simple, just use our notation. So when you describe an optimization problem, it should look like the way we do in notes, finals, lectures, all that stuff. So I guess they'll be due tomorrow at 5:00 or something like that. If you're still sort of in active discussions with us and haven't quite closed in on that, that's okay, but you should be – then come to us and let us know you're gonna do it over the weekend or something like that. And we'll consider this just the first cut. Are there any questions about that, about how that's gonna work? Anybody still looking for a project, because we can assign. Every lecture I say, "This would make an outstanding project." No one's ever gotten back to me – ever. We can make up project for you, too. Yeah?

Student: Can we changes to the initial proposal?

Instructor (Stephen Boyd):Yeah. Don't even think about handing us something that's 15 pages. We're not even – actually, ideal would be a half page. I mean, you have to have just the right project. By the way, you might be able to pull it off, half page. Okay. That's fine. That would be great. Shorter the better. And if there's any previous work or something like that, we haven't asked for it, but at some point, you're gonna have to do that. So I mean, I presume at this point, if you have an idea of a project, you've kind of done some Googling and all that. We assume that. That goes without saying. And you can add some references or something like that, if you want. Okay. Any other questions about this? Otherwise, we'll just continue. Oh – no, we'll wait. So last time we looked at localization methods. So the basic idea of a localization method – you can think of these as just generalizations of bisection. They're generalization to Rn. So you have bisection in multiple dimensions, and it's not a simple – there's no simple extension to Rn. Because in R, first of all, convex sets are kinda boring. They're basically just intervals. And there's an obvious center of an interval. In Rn, things are much more complicated. You have many different centers. I don't know if you have that point. Where is it? Here we go. Thanks. Okay. So the basic idea is you have some polyhedron, where you know the solution – if it exists. And you query a cutting-plane oracle at the point x, and what comes back – in this case, this is a neutral cut here. In other words, what it's basically telling you is that you need not even consider anything over here. So the solution is not here, if it exists. And so this is your posterior or updated ignorance set, or whatever you want to call it. It's the set of points or localization set. It's the set of points in which, if there is a solution, you know it must lie. I should mention one thing, there's more material on this, of course, in the notes, which you should be reading. And we're expecting everybody to be reading. Okay. So bisection on R is the most obvious and well-known example. Let's look at some other ones. We'll look at some of these in more detail today. The first is the CG method. I just got up to that at the end of last lecture. Here we have x(k+1) is the center of gravity. I'll review that quickly and say a few things about it that I didn't get to say last time. Other would be maximum volume ellipsoid, Chebyshev center, analytic center. These would be some of the other ones. And we'll talk about each of those in a little bit of detail. So the CG algorithm is the CG of the polyhedron of your query point. And there's an amazing fact.

In fact, we may even, if we can find a simple enough group, we'll stick it on Homework 4, which we're working on. And the amazing fact says this, that if you take any convex set in Rn, and take the center of gravity of that set, and then pass any plane through that point, the question is, how unevenly does it divide the volume? So anybody guess the answer's 50/50? For example, in R, it's 50/50. But it turns out immediately you draw a couple of pictures in R2 and you'll see that, in fact, there's no point – you can make a convex body for which there's no point for which all lines going through it divides the area 50/50. But it turns out you can't skew it too much. And it turns out than the actual ratio you can go to is 0.63/0.37. So it's roughly not even two to one. Because it's this divided by 0.37. Not even 2 to 1. It's more like 1.5 to 1. And that means, basically, if you find the CG of a set, and you put the worst hyperplane through that point, it will chop off at least 37 percent of the volume. So that's what it is. Okay. So that says in the CG algorithm, that's very nice, because it says the volume of the localized set goes down by the factor of 0.63 at least. It can go down more. If you have a deep cut, it goes down by more. And of course, the 0.63 is the factor for the absolute worst possible hyperplane at each step. And in fact, it obviously is typically much less, or something like that. So that's that. We'll look at the convergence and look through various things. And we came, at the last minute, to basically the problem with the algorithm. The problem with the algorithm is computing the CG is extremely difficult. So it's a conceptual problem is the way it's described. Now you can modify the CG method to work with approximately CG computations. I'm gonna mention some of those, because there's some very cool new work, not from Moscow, from MIT, and it's very, very cool stuff. I'll say some of that. Obviously, you don't have to compute the CG exactly. So first, there are methods even from the '60s, where a cheaply computable approximation of the CG is given, and then you have a chopping, a slicing inequity theorem, which is not as strong as the 0.63, but is enough to sorta give you convergence and so on. So those you already had in the '60s. So those are approximately CG methods. But let me just mention, actually, a really cool method developed four or five years ago, something like that, at MIT. Very, very cool. Goes like this. You want to calculate the CG of a convex set. And you can use a randomized method. And the randomized method is called hit and run. Has anyone heard of this? It's just very cool. It's an advanced class, so I can go off on a weird tangent and tell you about it. So the way it works is this, you have a point, and you generate a random direction. So you generate a random direction and you make a line. And you need the following: you need the ability to find out where that line pierces your set. So that's what you need. So here's your set. I'll make it a polyhedron, but it does not have to be a polyhedra. Just any old convex set. It doesn't really matter. Here's your set. And you take a point here, and you generate a random direction. Actually, how do you generate a random direction? Uniform on the unit sphere. How do you do that?

Student:[Inaudible].

Instructor (Stephen Boyd):Thank you for bringing that up. I should say, calculating the CG in 2D and 3D is no problem at all. 2D is trivial. All you do is you calculate, you find a point in the set, you calculate a triangulation of the polyhedron, done. In 3D, same thing. However, it's gonna scale sorta exponentially in the dimension. So if you're interested in 2D and 3D, CG is a non-problem. Okay. So how do you generate a random direction, uniform on the unit sphere? Just a random variable uniform on the unit sphere?

Student: Choose a random vector.

Instructor (Stephen Boyd):Yeah, fine, but how do you choose a random vector? What distribution?

Student:[Inaudible].

Instructor (Stephen Boyd):Why, because you can't think of any other? You happen to be right. You choose a random Gaussian vector. So it's n0i vector. That's circularly symmetric. And so if you then normalize that, you get a uniform distribution on the unit sphere. So you choose a random direction here, and so here's the line you get. And the only thing you need to do is to calculate these two intersections, which is very easy to do. Now you do the following: you generate a random variable uniform on this interval, and that is your new point. So that's gonna be right here. And you repast. So you go here, you calculate a random direction – the direction turns out to be this – you do a random point on there when you get there, and you go on. So I suppose these are the hitting points and this is the running, or who knows. I don't know how that name came up, but that's what it is. Everybody got this? So that's a Markov chain. It's a Markov chain propagating in the polyhedron. And it actually converges to a steady state distribution that's uniform on the polyhedron. So that's a method for generating a random variable that's uniform. By the way, if you have a method that calculates a random variable that's uniform on the set, how do you find the CG?

Student:[Inaudible].

Instructor (Stephen Boyd):Thank you. You just average. Let me ask you another question. Anybody got other ideas for how to generate a random point in a polyhedron uniform distribution?

Student: Run the distribution everywhere and reject the ones at the outside of the set.

Instructor (Stephen Boyd):It couldn't be everywhere, because it has to be uniform. Okay. So what you do is you find a bounding box. Everybody knows how to find a bounding box. How do you find a bounding box for a polyhedron? This is the problem with scheduling class at 9:30. It's before full outer cortical activity has warmed up. How do you find the bounding box for a polyhedron?

Student: Minimize x1 [inaudible]. Instructor.

So you minimize x1 and you maximize x1. You minimize x2 and you maximize x2. And these are like LPs or something. Of course, maybe we're trying to solve an LP. I don't know what we're trying to do. But still, that's how you do it, by solving two NLPs. So you make a big bounding box like this, and now you generate a uniform distribution on that. And you do that by taking a uniform distribution of each edge and sticking them together. And then you reject the ones that are outside this thing. And the ones inside, actually, then, are drawn from a uniform distribution on the polyhedron. Any comments on how you imagine that might scale with the dimension? I'm not saying you have the bounding box.

Student: Exponentially.

Instructor (Stephen Boyd):Which way, though? Exponentially what? What would happen exponentially? You're right, something happens exponentially. What is it?

Student: To test if the point is in the polyhedron?

Instructor (Stephen Boyd):No, no. To test if a point is in the polyhedron, you form ax. You give me x, I calculate ax. I check if it's less than or equal to b. That's fast. What happens.

Student: Volumes.

Instructor (Stephen Boyd):There you go. Ratio volumes in the worst case goes down exponentially. And that means you'll generate exponentially many random points before you even get one in this set. So although that does, indeed, produce a uniform distribution on the set, it's not gonna scale gracefully to the high dimensions. Okay. So anyway, this is the hit and run method. There's lot of – you can certainly do a cutting-plane method with something like this. And a more sophisticated version. Okay, so that's the CG method. Yeah?

Student:[Inaudible] in the vertices, where it intersects?

Instructor (Stephen Boyd):Oh, here? Oh, that's actually quite straightforward. If this is a polyhedron, let me show you how that calculation goes. So the calculation goes like this: here's your polyhedron. It's defined by that inequality. So ax less than b. You're given a base point x0 and a direction, v. And you want to find the maximum and minimum values for which this holds. That holds for t for an interval, right. So I'm assuming this is bounded polyhedron. I'm assuming x0 is in the polyhedron, although none of that matters. So that says that a(x0), that's if t equals zero, is less than b. And now you want to find how much can you increase t until this becomes false. And how much can you decrease t below zero until it becomes false. That's the question. So you just write it this way. It's actually kinda dumb. You write a(x0) plus t times av is less than or equal to b. That's a vector, that's a vector, now it's very easy to work out the maximum. Because this is basically – let's call this a plus t – no, no, that wasn't a good idea. How about a plus tc is less than b. Now it's gonna work. Now it's extremely easy. What you have to do is this, if I increase t, this thing – by the way, if a ci is negative, or zero, increasing t doesn't hurt – we're not going to get into trouble there. So if I want to know how high can I increase t, I only focus on the positive entries of c. So I first look at the positive entries of c, and I divide them by a, or something like that, and then I take them in, or something, and that's my – well, I do something. Anyway, it's a 300-level class. I can just say it's easy to do. This made sense, though? So in fact, when you actually do this, it's two little lines of something. Okay. Maximum volume ellipsoid method, this method is not a conceptual method; it's a real one. Actually works really well. And here, x(k+1) is the center of the maximum volume ellipsoid in Pk, which, I don't know if you remember or not, but that can actually be computed by solving a convex problem. By the way, how about the minimum volume ellipsoid method? In that case, you calculate the minimum volume ellipsoid that covers the polyhedron. Let's have a short discussion talking about the minimum volume ellipsoid method. I don't know if you remember this, but you can kinda guess. Any comments on the minimum volume ellipsoid method? By the way, it would work really well. You would get a guaranteed volume reduction and everything.

Student:

[Inaudible].

Instructor (Stephen Boyd):Yeah. In fact, finding the minimum volume ellipsoid that covers a polyhedron defined by linear inequalities is actually hard. The method is like the CG method. I gave you enough hints that you could figure it out. Did you actually remember that?

Student:[Inaudible].

Instructor (Stephen Boyd):Oh, and then your remembered. Good. Okay, that's a short discussion of the minimum volume ellipsoid method. But the maximum volume ellipsoid method is actually a quite – it's not a conceptual method. It actually works very, very well. And in this case, the volume reduction you get is actually very interesting. And the volume reduction is this: the guaranteed volume reduction is not a fixed number, as it is in the CG method. In the CG method, you will reduce by 37 percent, end of story. Any dimension. In this case, your volume reduction guarantee degrades with dimension. By the way, it degrades gracefully enough that this method can actually be used to construct a polynomial time convex optimization method. Because a good enough approximation of the maximum volume ellipsoid can be calculated in polynomial time and so on and so forth. And this is also called – it's got four initials, TK – Russian names go together. I won't attempt it. Actually, I think it's described in the notes. Now, once you know to get this volume reduction, you can work out a bound on the number of steps required, and in this case, it scales not like n, as it was before, but it's actually n2. So this one degrades. And this is actually quite a good practical method. Okay. Other ones? You can pretty much make up your own method here. Chebyshev center method, I thought I'd mention it. Now, the problem with the Chebyshev center, that's the largest Euclidean ball in Pk, and that's solved by an LP, by the way. That's a linear program to calculate the maximum volume ball that fits inside an ellipsoid is just an LP. Now this one, however, is not affine in variant. In other words, if I take my original problem, and I do a linear change – if I do a scale, actually, if I scale the variables, maximum volume ellipsoid is CG, you get a commutative diagram. In other words, if you take a polyhedron and you subject it to an affine mapping, you multiply it by t and add some s, or something – t's a non-singular square matrix – then the CG will also change coordinates exactly the same way. Because CG commutes with an affine change of coordinates. So that means that the CG method is actually affine coordinate change in variant. Okay. The same is true of maximum volume ellipsoid. If I give you a set and I ask you to find the maximum volume ellipsoid inside it, if I then change coordinates by an affine change of coordinates and ask you to do it again, it commutes. So the maximum volume ellipsoid that fits in the transform thing is the transform of the maximum volume ellipsoid that fit in it. So that was my verbal way of drawing a commutative diagram. Now, this has a number of implications. First of all, for our complexity theorist friends, this is, of course, extremely important.

But it actually has lots of practical implications, and they're quite serious. Basically, you have a method that's affine in variant, in practice what it means is scaling of variables is a second-order issue. Now in exact arithmetic, of course, it's a non-issue. But it means it's a second-order issue. It's a second-order issue because scaling can hurt you, but only through numerical round off and things like that. Therefore, being off by a factor of 1,000 in the scaling coordinates and no one gets hurt. Whereas, a method like Chebyshev center, here, I guarantee you, if you have your Chebyshev method, the center method works really well, and I scale, for example, the first and the eighth coordinate by a factor of 1,000 to 1, your method is not going to work, basically at all. So that's the basic point there. Now having said that, I should say this, if you're solving a specific problem where you know what x1 is, x1 is measured in meters per second, and it varies between plus/minus five. And x7 is a pitch rate in radiants per second, or something like that, or radiants per second squared. And you know how it's scaled. So in actual specific practical problems, the whole business of affine and variants may not be that relevant, because in a specific problem, you generally know what the numbers are, what the ranges are and things like that. Okay. Now we're gonna look at this one in considerable detail, actually, in the next lecture, because it's one that this one seems to have a very nice property. It actually works really well in practice. And it also has some nice theoretical properties. So in this case, you take x(k+1) to be the analytic center of Pk. I should mention that this English here, the analytic center of Pk, that's slang. That's informal and it's actually not correct. Because the analytic center is not the analytic center of a geometrical set. You have the analytic center of a set of inequalities. And obviously, you can have multiple sets of inequalities that describe the same geometric set. And in fact, take your favorite polyhedron, and add all sorts of redundant inequalities. The set hasn't changed; the analytic center has. And in fact, the following is true: take a polyhedron, and take any point in the interior of your polyhedron. So pick any point at all. It can't be on the boundary, but take any point in the interior of the polyhedron. By adding redundant inequalities, you can make any point you like the analytic center. Now, you may have to add a large number of inequalities and things like that, that's true. But the point is, you should remember that this is informal and it is slang. So meaning it's just not right. Okay. So the analytic center is this. It's the argmin, of course, of the sum of the logs of – you maximize the sum of the logs of the slacks. These are the slacks in your inequalities. By the way, this is the same as maximizing the product of the inequalities. So it's exactly the same thing. You maximize the product of the inequalities. By the way, if you – there's many other things you could do. You could do things like maximize the minimum slack.

And if the ai's were all normalized to have the same length, for example one, that would correspond exactly to the Chebyshev center. Because bi minus ai transpose x is norm – whatever it is. It's the distance to that hyperplane divided by the 2 norm of ai. So this works quite well. And we're gonna go into that in great detail later today. Okay. For these cutting-plane methods, there are lots of extensions. I'll mention a bunch of them, and when we look at center cutting-plane method, we'll look at how these things work. So once you get the idea that the only interface is a cutting-place oracle, then you can imagine all sorts of cool things. So you could do all sorts of things. You could do things like this. Your oracle, when you call your oracle with x, instead of just giving a single – I mean, there's tons of stuff you could do. Instead of just returning a single cutting plane, you could easily imagine something that returns multiple cutting planes. And I'll give you an example. Suppose you're solving an inequality constraint problem. The simple method would be this: you take your constraint functions and you evaluate each one. You have to do that. So you evaluate f1, f2, f3, and the first time you find an f that's positive, you call that f, fk.getsubgradient. And you get a subgradient back. That's one method for generating a cutting place. But in fact, what you could do is if you sweep all the constraints, you could then get the most violated constraint. But another option is to give all the constraints, just give them all at once. And you just update your polyhedron and everything's fine. By the way, you can actually return not just – if you return multiple inequalities, you could actually return shallow cuts. Shallow cuts are fine, as long as one of the cuts you return is actually deep, or neutral. So it's still useful information. So let me just draw a picture to show what that would look like. So that would look something like this. And you might get some other kind of very colorful name for it. Here's your localization set. Here's a point – actually, it doesn't matter however you calculated it, CG, hit and run, analytic center – it doesn't matter. What would happen is you call it here, and you could do something like this. It could give you a deep cut, basically says, don't bother looking over here. But there'd be no reason that it couldn't also give you a shallow cut like that, that says, don't bother looking here. No problem. So you just incorporate both. So that's how that works. I mean, that's kinda obvious. Actually, what's interesting about it is that a lot of these things don't really make it work much better. Okay. Instead of appending one, you append a set of new inequalities. You can also do non-linear cuts. So you'll also find this talked about all over the place. And let's see, these would work – you could have quadratic cuts, where you return – that basically says, instead of returning a half space, where the solution, if it exists, must lie, you'd return an ellipsoid, basically. And there'd be plenty of cases where something like this would work, if your inequalities were quadratic or something like that. In this case, the localization set is no longer a polyhedron, but that doesn't really matter. I mean, in particular, things like ACCPM, they work just fine.

So these are kind of obvious and the ones, however, that are quite useful, and also quite strange and mysterious, is the idea of dropping constraints. So dropping constraints would go like this: after a while, you generate a big pile of constraints. So what happened was, you started with a whole bunch of inequalities. At each step, you have added at least one inequality to your set. So what's happening is the polyhedron, the data structure that describes your current ignorance, is growing in size as you go. And that means, for example, whatever work you're gonna do on it is also gonna grow in size. I mean, like grow linearly or polyhedrally – sorry, like polynomially. But at least it's gonna grow linearly. When you add these, there's the option of deleting dropping constraints. So then how do you drop constraints? The most obvious this is the following: here's your current set. And here are some redundant inequalities. So one obvious thing is to call a method on your current polyhedron, which is minimal. And what minimal does is it goes through every inequality, and drops the ones that are redundant. By the way, how do you find a redundant inequality? Suppose I give you a polyhedron and I point to the 15th constraint and I ask you, "Is it redundant?" How do you do that? How about that first one? How do I know if the first inequality is redundant in a polyhedron? Got any ideas?

Student: If it's linearly dependent on other columns?

Instructor (Stephen Boyd):No, that might be wrong, because the b it connects into. But that would be one. That's how you detect linear – that's linear equality equations being dependent. But that's – it beats relatedness, but there's actually really only one real way to do this.

Student: Consult an LP and see if it's strictly feasible?

Instructor (Stephen Boyd):You got it. Here's what you do. You want to know if a1 transpose x less than bi is redundant. Actually, there's a very sophisticated way to do it, but this is very simple, basic, everyone understands it. Here's what you do. You maximize a1 transpose x over the remaining inequalities. That's an LP. Now, the optimal value of that LP is either less than or bigger than b1. If the maximum of this over b1 is less than or equal to b1, what can you say about the inequality? Redundant. So that's one way to do it. Otherwise, actually, what that method will do is it will either produce a point that satisfies all the other inequalities, but not that one, or it will produce, basically, a certificate proving that inequality's redundant. And actually, if we get back to your suggestion about linear dependents and stuff like that, there's more sophisticated ways. In fact, is it relevant? I don't know. That's a great homework problem in 364a, but I guess this isn't 364a. Anyway, the question would be, how do you find out if the first five inequalities are redundant. You should think about that. Okay. You could implement a method that was called polyhedron.reduce, and it would just go through and solve one LP for each constraint. So it would check an LP for each constraint and drop that. Now, if, by the way, the subgradient calls are fantastically expensive, the subgradient calls are cutting-plane oracle calls. If the cutting-plane oracle calls require a network of computers to fire up and the wind tunnel at NASA aims to get fired up and all that kind of stuff, then no problem calling localization polyhedron.reduce. No problem. But in a lot of cases, that would actually be too much – to solve whatever k LPs would actually – anyway, okay. There's gonna be cheap ways to drop constraints, and there's safe and unsafe constraint dropping. Safe constraint dropping goes like this: you drop constraints that, without actually solving an LP, you know are going to be redundant. We'll see how that's done with analytics and cutting plane. Very cheaply. That's one method.

And the other method is the unsafe method, is you just drop constraints, whether or not they are redundant. That method works, by the way, really well. Capital N here. It seems sort of like the word on the streets is that if capital N is five little n, everything's cool. There are some proofs of this and all that, but they're very complicated, as you might imagine. When you are dropping constraints that are not redundant, you are increasing the volume of that set. That's our merit function. So now the proofs get very tricky. And you have to show that your constraint dropping method is increasing the volume of the localization set more slowly or roughly than your cutting planes are reducing it. So that would be it. But the fact is that we'll see something like N equals 3 to 5 n works, apparently, very well. Okay. In fact, there's a shock. We'll see it next lecture, which is gonna come up very soon. What's weird is you can actually drop constraints, and the method will do better. I know it's a bit weird, but it's also true. It can happen. Okay.

Another variation on this is so-called epigraph cutting-plane method. If you have – it works like this. You put the problem into epigraph form like this, and what you're really going to do is get a cutting-plane oracle for this problem here. So this is the problem. And what you want to do is you want to separate x and t. So x should be considered as your current proposal for x, or something like that. And t is to be considered here and upper bound on the objective. And you're gonna separate this from x*, p* where p* is the optimal value. Now the way to do that is this, if the current point is infeasible for the original problem, and violated the jth constraint, you add a cutting plane that is associated with that violated constraint. And this is a deep cut here. I guess this is positive by definition. This is a deep cut here, this one, where you get a subgradient of the violated constraint here. And this is a deep cut here. And you can also just put this with zero here and that would be a neutral cut for the same problem.

Now, if x(k) if feasible for the original problem, you can actually add two cutting planes. And the two cutting planes you can add are this. This is a deep cut here. This is a deep cut that you add in x and t here. And this here is actually something that says that the current point, that's the objective value because feasible, t, which is supposed to be an upper bound on the optimal value, is clearly less than that, so you can add that one, as well. And this requires getting a subgradient of the objective. So that's how you do this in a – you can run a cutting-plane method in the epigraph. Okay. Now you can also get a lower bound. We'll get to some of these in a minute. This is related to something called Kelley's method. But suppose you evaluated a function, f, and a subgradient of f at q points. So it doesn't need to be convex, but of course, it has to have subgradient. What that means is that f(z), by definition, means f(z) is bigger than this, all z. So it says that this is an affine global lower estimate on f. Well obviously if f(z) is bigger than each of these, it's bigger than the max. So you can write it this way. And this piecewise linear function here, we're going to call that f hat. So it's a very useful way to think of what it means if you get a subgradient at a bunch of different points, a single subgradient will give you an affine lower bound. If you call multiple subgradients, if you get multiple subgradients at different points, for example, if its an algorithm and your k steps in, and at each step you've evaluated subgradient, you actually have a piecewise linear lower bound. And that's this thing. And I'll draw that. It's kinda obvious what it is. But let me just draw it just to give you the picture of this. So here's the picture of that, other than R, which is kind of a stupid example. Here's your function. And let's suppose I've evaluated the – radiant, sort of here, you know, and here, and here. And what you do is you simply draw these first-order approximations, like this. There you go. Something like that. And then this one looks like that. And this function right here that goes like this, that's a piecewise linear lower bound on your function. By the way, if you minimize that, you get this point here, and that's clearly a lower bound on your function. The point is that once you evaluated a couple of – not a couple, but in fact, n + 1 subgradients of a function in Rn – if you evaluated n + 1 subgradients in Rn, then generically, you can now produce a lower bound on that function. The cost is solving an LP. So what happens is this, if you replace f0 and fi by their polyhedral under-approximators, then solve this problem, which is an LP. Well, it's not an LP, but it can be converted to an LP. You know how to do that.

So if you convert this to an LP and solve it, I guess this one's trivial, and this one you introduce one epigraph variable t, you get an LP. You solve that and the optimal value is a lower bound. In fact, it's this value here. By the way, we didn't mention this. I'll mention it here, because it's an interesting method. If you solve that LP here, then you get a lower bound on the optimal value, and if this is your new point at which to query, you get a nice and you get a famous algorithm. So if your next query point is here, and it goes like that, that's called Kelley's cutting-plane method. And it's actually quite a good method from the '50s, I think. Either '50s or '60s, one or the other. That's Kelley's cutting-plane method. And what would happen is when you add this, your lower bound goes up there. And your next point in here. And you can actually see in this case it's a smooth problem, that it's gonna do pretty well. It's not hard to show that it converges and so on. A small modification of Kelley's cutting plane method, actually, makes it work really well in practice, and it's not clear from this picture why it wouldn't. But of course, pictures are generally in R2 and R3. And that's not where – there's no interesting problems. These aren't real optimization problems. Real optimization problems go down in like R10 minimum. They get mildly interesting in R100. And actually are real honest problems in R100K, or something like that, where your intuition in drawing pictures isn't going to cut it even close. Okay. Let's see. So we'll go on and do the analytic ACCPM. Okay. So we'll look at ACCPM. I think this is sort of – I'm not sure, but I guess I'd probably put this at the if someone needed to run one of these non-differentiable methods or something like that, a cutting-plane oracle, or something, and you actually really needed to do this, I think this is probably – depends on the situation – but this is probably the algorithm of choice. We'll look at that. And there's a couple of things we have to talk about. So the first is, is just to remind you what the analytic center is.

Again, this is not correct. Oh, unless – well, this is wrong. This notation universally defines p to be this set, and this is not a function of p. It's really a function of the inequalities. But we'll leave this and just understand that that's slang. So what you have to do is you have to maximize the product of the slacks in a polyhedron. And that can be done by Newton's method. This is completely smooth. It's actually self-important. So that means if you're gonna give a complexity theory result for it, you're already well on your way. And here's the ACCPM algorithm. You start with a polyhedron. By the way, it doesn't have to be even a polyhedron, but it doesn't matter. You start with a polyhedron known to contain the target set. Then you go this, you calculate the analytic center, you query the cutting-plane oracle. The cutting-plane oracle can tell you that you're done, in which case you quit. Otherwise, it returns a cutting-plane inequality, that's this, and that is appended to p, the description of p. Now if this polyhedron is empty, then you can announce that capital X, the target set, is empty and quit. Otherwise, you increment k and repeat. So that's ACCPM. Let's see, for constructing cutting planes, I think we've actually looked at a bunch of this before, but we'll go over it anyway. If you have an inequality constrain problem, then you do the following: if x is not feasible, you choose any violated constraint and do an added deep cut. If it's feasible, you can do a deep objective cut. And the deep objective cut is given by you maintain the best value you've found so far, and you add this. So that's a deep objective cut. You get the analytic center. You have to solve this, and there's lots of ways to do that. But there is one issue, and that's this. You're not given a point in the domain of this function. So that's actually the challenge here. And there's lots of ways to get around this. One, of course, is to use a phase one method to find a point in the domain. And that actually will have the advantage here of if you do that, the advantage here is going to be that that will surely, in a graceful way, determine if this thing is empty. If it's not empty, you can use a standard newton Method to get the analytic center. You don't have to use an infeasible start Newton method. And there's one more method, which is you can actually solve dual. You do the dual of this analytic centering problem. And if you think about why that will work very nicely, you have a set of inequalities, the dual has a single variable for each inequality. If you add an inequality, you're actually adding a new variable to the dual. And so, there the initialization is easier.

So that's yet another method. Infeasible start Newton method, actually, the truth is I didn't cover it, really, at all in 364a. It's in the book and stuff like that. But here it is. I covered it maybe too fast. You want to minimize this separable convex function subject to y equal b minus ax, so we introduce this new variable, y. And the way it's gonna work is this: the infeasible start Newton method does not require the points to be feasible. The traditional Newton method you must start with a feasible point and every direction you generate is in the null space of a, and so the points remain feasible. So it's a feasible method. Infeasible start Newton method, you don't have to be feasible. So here's the way you would solve this problem. You initialize x as anything you like, zero, the last x you found, some guess, or something like that. And a good – the radical method is to initialize all these y's to be, for example, one. Why not? That's a point nice and squarely in the domain of the objective, which is the sum of logs. When you do that, obviously, this equality constraint is not satisfied, unless you're really, really lucky. Unless it turns out that your x. So this is the initialization. So here would be a common choice. Suppose you had to guess x previous. You could choose y to be the following: you would evaluate each bi minus ai transpose x, that's the ith row here, and if that's positive, no harm. Leave yi to be equal to that. If it's positive, it's in the domain of log y. And there might be a little threshold value, but small enough you don't. Otherwise, you just set yi to be one. So what happens, then, is these equality constraints here are some are already satisfied. But anyone where this is not the case are not satisfied. So that's how that works. By the way, this infeasible start Newton method is very cool. It has the property that if any constraint, literally coordinate by coordinate, is satisfied at any step, it will be satisfied forever afterward. So for example, if there's 100 inequalities here and 80 of these are satisfied immediately, but 20 are violated – or let's make it even simpler.

Let's do cutting-plane method. So in a cutting-plane method, you just solve the problem with 99 inequalities. You added one inequality, and now you have 100. You satisfied all the 99, because you were actually at the analytic center of those 99. However, the new constraint you add, you for sure violated. Well, you might just have equal to zero. It might be just – if it's a neutral cut. If it's a deep cut, you're guaranteed to violate that point. But then what's cool about that is only that one – you have 99 equality constraints here are satisfied and only one is not, which is the new one. Everybody understand? Okay. So the method works like this, you define the primal dual, the primal residual is y plus ax minus b. Now of course, in standard Newton method, you are always primal feasible. So you don't even have the concept of Rp. Rp is just zero always, initially, every step Rp is zero. And it stays Rp because your Newton step is in the null space of a. So no matter what step you take in the null space of a, you continue to satisfy ax equals b. In the infeasible start Newton method, that's false. Rp starts off – in fact, it's guaranteed of the original point was not feasible, starts off non-zero. And the dual residual is a transpose nu ng plus b plus nu. So these are the components associated with both x and also with y. Okay. And here the gradient is very simple to calculate. It's simply the vector, which is basically one dot with a y minus sign. That's the gradient. Now the Newton step is defined as follows: you actually linearize these equations at the current point. And you'd linearize the equations Rp equals zero and Rd equals zero. Now, this equation is already linear, so there's hardly any linearization going on there. This one is not linear, because g is a non-linear function of y. And you'll get something that looks like this. Now, h here is the heshen of the objective, so that's diagonal. You get a and you get a transpose and so on. Now that's a large system. The system is size, I guess, let's see, I can't remember the dimensions of n and m, but y is going to be m, so that m by m – this is not a small system here. It's quite big. However, this should be screaming structure at you. Screaming at you with structure. There's tons of zero's, there's i's. This is diagonal. There's a and there's a transpose. So you should have an overwhelming and uncontrollable urge to apply a smart method to solve these equations. And in fact, you can do this by eliminating various blocks and things like that. Instead of having a complexity, which is cubic and m plus n, you'll actually do much better.

What'll happen is, you can actually solve them by calculating delta x this way and delta y that way, and so on. Now when you calculate delta x, you have a couple of different methods for actually computing delta x. One is you actually form this matrix. And in fact, here you can exploit any structure in a to form this matrix. But this matrix, you can – even if the matrix is dense, you can exploit structure informing it. That would be one method. If this multiplied out matrix here, if this heshen is actually has structure, you could use a Sparse Cholesky Factorization or dense or something like that. And the other method for solving this, and this would be probably the method that would be not frowned upon by people who do numerical things, if you just simply worked out what this is, this delta x is the solution of this least squares problem. If you actually form this matrix, a transpose ha, I guess that's a – most people who do numerical things would consider that a misdemeanor. It's called squaring up. So whereas, over here, you never actually form this matrix, which has, whatever, the square of the condition number of this thing here. So this is another way to do it. And by the way, the h halves don't have to scare you, because they're actually diagonal. And this is nothing but a diagonal least squares problem. So here's the infeasible start Newton method. You start with a point, x, arbitrary. And y has to be positive. It could be all ones or that initialization we talked about before. You have your usual parameters. And what you do is you calculate the Newton step in delta x, delta y, and delta nu. And you do a backtracking line search. In infeasible start Newton method, you cannot use the function value to do a line search. That's completely obvious because you're infeasible, so your objective, actually, can be very, very good. Very good. It's just totally irrelevant because you're not even feasible.

And you really can't – anyway, it would make no sense. And in fact, you could easily see the method would fail, because you initialize it at any point that's infeasible, but has an objective value that is better than the optimal objective value, which of course, is higher because it requires feasibility. Then if you did a line search or just a numeric function, the function value you'd never get to the optimal point. That would pay off. Instead, what you do is you do a line search on the norm of the residual. And that's easy enough to do. The simple way is something like y plus t delta y is not positive, you could do this. By the way, related to the question you asked earlier about this, it's actually must more common. In fact – it doesn't matter. You can either t time equal beta in a while loop here, and simply find out if you're positive. But the fact is, you'd probably immediately, if you're writing this in a real language, what you'd really do is you'd go here, you'd get y and you'd get delta y, and you'd quickly calculate the largest t for which this is still the case. And then you'd take the minimum of that t multiplied by 0.99 and 1. So that'd be your minimum. And then you'd check. You'd drop down to this one. And then you'd t times equal beta over here until this is satisfied. And then you update. Now there's actually some pretty cool things about this method. This method has several properties. I'll just mention them and you will see them at some point soon. Maybe on Homework 5. I don't know. You'll see them. One interesting point about it is this, if you ever take a step size of one, then all equality constraints become satisfied on the next iterate. And they will always be satisfied from then on. That's one property. Another one is if any of the individual equality constraints is satisfied, ever, at some step, it will be satisfied forever on.

Actually, that's unlikely to occur during the algorithm, so probably the better way to say that is something like this, any equality constraints initially satisfied will remain satisfied during the algorithm. Then there will be a point at which you take a step size of one, after which all equality constraints will be satisfied. That's how that works. Okay. I think I already said all these things. There's a couple things to mention. This method does not work gracefully if there is no feasible point. Feasible means there's no point in the domain. So there's no positive y for which y equals ax minus b, or something like that – b minus ax, y is b minus ax. There's no positive y that satisfies that. And then, of course, obviously, what happens is the Newton method churns along. It obviously cannot take a step length of one, because if it took a step length of one, the next would be feasible, but that's impossible, so it doesn't. So there are methods you can do to recognize infeasibility and stuff like that, but I think the best way to say it is that infeasible Newton method is designed for things, which are going to be feasible. And they just don't gracefully handle infeasible things. Let's go back to this pruning and I can say something about this. This is also material from 364a and in the book, but it's a big book and 364a we went way fast, so I didn't expect anybody to get it. So we're just doing backfill now. These are actually very, very good things to know. So if you have the analytic center of a polyhedron – again, that's slang – then if you calculate the heshen at the analytic center here, and that simply looks like this. It's nothing but this thing here. The gradient, of course, of the analytic center is zero, obviously. That's the definition of it. Then it turns out the following is true: that defines an ellipsoid. In fact, let me say something about the other one. Because there's actually some very good things to know about these. These are in the book, I think, but I can't really remember. Here it is. Here's a polyhedron. Here's a point, which is not the analytic center. Calculate the heshen, that's this thing up here. The heshen of the log barrier like this, like that. That's not the analytic center. And this ellipsoid here is the set of x such that x minus z transpose h, this is less than one. So you'd simply calculate that heshen right there, that formula, without the stars. You calculate that heshen. This ellipsoid is guaranteed to be inside that set, always, whether or not you're at the analytic center. That's gonna come up later, because we're gonna look at another interesting method, very interesting method called the Deacon method.

So at any point, if you calculate the heshen of the log barrier function, then the ellipsoid with a one, defined by that heshen, is inside this set. Period. Let's go to the analytic center. If you go to the analytic center, let's say that's here. Then this result holds, because it actually holds for any point. So if I draw the ellipsoid like that – I made it too small, by the way. You'll see why in a second. If I draw it with a one, it's inside the set. And what this says is if I draw it with an m2 – let's see, one, two, three – how many of these? Six? So if I draw this ellipsoid with a one, I'm inside it. This result says if I puff that up by a factor of m, which is six in this case, you will cover the set. No, I'm gonna miss it. I didn't draw it big enough. But theoretically, if I drew this right, and I puffed up that set by a factor of m, I would cover the set. So if you calculate the analytic center, you get, actually, an ellipsoidal approximation of your polyhedron. And it's an ellipsoidal approximation that is within sort of a factor of m in radius. In other words, you have an inner ellipsoid, you have an ellipsoid guaranteed to be inside. Puff it up by a factor of m, guaranteed to be outside. By the way, the fact that this is guaranteed to cover the set is not – that's actually interesting, because if I just give you a polyhedron, and I give you an ellipsoid, and I say, "Would you mind checking if your polyhedron is inside my ellipsoid," that's NP hard, because you have to maximize a convex quadratic function over a polyhedron. So you can't even check if a general one was. So in this case, there's a fast way to verify that this is true. I think that this is easy to show. It might even be – you may even do it yourself. Who knows? So what this allows you to do, now – and you're gonna calculate this if you use analytic center cutting-plane method, you had to calculate this anyway, because that's that ah a transpose. That's all that is.

So you've already calculated all this stuff, anyway. And it turns out that you can now define something, which is this irrelevance measure. And it's this: it's bi minus ai transpose x* divided by this thing. Here's what you can guarantee now. It's actually the normalized distance from the hyperplane to the outer ellipsoid. And I guess I'm not ever sure I need the m here. Is that right? No, I guess you do. Maybe we should make eta go between zero and one. Anyway, I don't know. So here's what you're guaranteed. If eta is bigger than m, then the constraint is redundant, and that you know without solving an LP. And by the way, it's super cheap, because the fact is, I think you've actually already calculated at the last step of Newton's method, I think you've actually already calculated all these numbers. So I think you have, but we can check. But I'm pretty sure you've already calculated all these numbers anyway. Or you're very, very close to getting them. Well, we'll leave it. You have, because you've actually factored – it's cheap, anyway. You've factored this to calculate the last Newton step. So again, if you're doing direct methods, you've factored that, so once you've factored that, computing this is actually the norm, you have a Cholesky factor of that, this is the norm of L minus 1 ai and that's one back substitution.

Student:

[Inaudible].

Instructor (Stephen Boyd):Hang on here. If you have a whole bunch of constraints in describing P, the analytic center cutting-plane method says you've gotta calculate that heshen. You have to calculate a Newton step to calculate the analytic center. So that's done. You're gonna already be doing that. Now what is true is that there's no reason to believe that any of – this is safe constraint dropping. In other words, if you drop a constraint based on eta being bigger than m, it's safe. You don't have to avoid your complexity theorist friends and things like that. The proof is still very, very simple, because the pruning does not change the set. It changes the description of the set, because you dropped irrelevant inequalities, but it does not change the set. So you can hold your head high if someone grills you and says, "Are you absolutely certain this method will converge?" Then you can say, "Yes." So that's safe. There's no reason to believe, actually that this will work. So in fact, what's generally done is you keep 3n to 5n constraints, and you just keep them in order by this. So you drop the ones with the – you keep the 3 to 5 n constraints with the smallest values of etai. Now, I can describe it another way. Let me describe it another way. Here's another way to describe this method. The method goes like this, and in fact, it's a perfectly good way to describe it. It goes like this: here's your polyhedron like that. And you calculate the analytic center. Now, you change coordinates so that the heshen of the log barrier at that point is i. So in other words, what that does it you make your set – you're rounding your set. You're making it more isotropic. Now, what you know now is that a ball of radius one fits inside your set. But a ball of radium m is outside your set, covers your set. So that's what's happened. And so now, the irrelevance measure is merely the length in those transformed coordinates. So you can say all sorts of interesting things about it.

Actually, it says that no value of eta could possible be less than one. It's impossible. Because at that center, the ball of radius one fits inside your set, and eta is the distance to the first hyperplane. So all the eta are bigger than one. Any of the eta's bigger than m are guaranteed to be redundant. That's the picture in the transformed coordinates. Okay. And you get a piecewise linear lower bound. We can go through that. In ACCPM you get that trivially, because you get the subgradient. You can form the piecewise linear approximations. We know this. And you can actually get this piecewise linear lower bound in analytic center cutting-plane method. That's not surprising because the analytic center cutting-plane method calculates an analytic center, but an analytic center is a step and a barrier method. When you finish a barrier method, you get a dual feasible point, which we know. That's the basic idea of a barrier method. And so it's not surprising that you're gonna get a lower bound. The details don't matter, I think. But the point is that this x(k+1), that's the analytic center of the big thing that's gonna satisfy something like this. And this is maybe better done in the notes, rather than in slides. Anyway, you end up with a lower bound from all the stuff you just calculated. And it's based on the idea that you calculated an analytic center, analytic centers are associated with center path, on a central path you've actually calculated dual feasible points, whether you know it or not. Dual feasible points means you have lower bounds on the problem. And so when you put all that together, that's the lower bound right there that you've actually calculated. Okay. And ACCPM isn't a decent method. So you keep track of the best point found and so on. Let's just take a look and see how this works. So here's a problem instance with 20 variables, 100 terms. So I guess it's a piecewise linear function. I think it's the one we've been using all along. So this is what it is. And I guess the optimal value's around one. So that means 1 percent accuracy. That means 0.1 percent and that means like coordinates of accuracy. And this is f minus f*. Obviously, you don't know this at the time. In this case, it's an LP, and you solve the LP to find f*. And then this shows you the progress. And you and see that it's not a – we can say a lot of things about this plot if you compare it to the subgradient plots. This appears to be, and that's actually correct, so they're proofs of polynomial time converge and everything like that. This appears to be what we'll call linear convergence, because that's a log scale and there's like a line that looks like that.

And that means something like this. That roughly each iteration gives you a constant factor of improvement. You can work it out by calculating the slope there. Now, subgradient is very different. Subgradient you're talking like one over square root of k or slower convergence. And that doesn't look like this. It looks like that, and then it gets very, very, very slow. Okay. This is actually, if you look at just the best point you found so far. We'll come back and look at some of these later. And this shows you the real effort. The real effort is the number of Newton steps required. And in fact, you can see here that the infeasible start Newton method at some point jams. So each of these is like a constant here, and the width of a tread is the number of Newton steps required to find a feasible point and then calculate the analytic center. By the way, things like this, these are just appalling. That's 200 Newton steps or something like that. So this is sort of the amateur implementation. The correct method of dealing with this, which would have made this thing go like that, would have used something like a phase one there. So phase one would be like far superior. This is just to show. And then this shows the bound and everything like that. We're out of time, so we'll come back and get to this later, although I will just show one more thing, which is this, and we'll talk about it next time. If you drop constraints, you do very well. Okay. We'll just continue here next time.

[End of Audio]

Duration: 78 minutes

ConvexOptimizationII-Lecture07

Instructor (Stephen Boyd):Well, this is – we're all supposed to pretend it's Tuesday. I'll be in Washington, unfortunately. So today I'll finish up and give a wrap up on the Analytic Center Cutting-Plane Method, and then we'll move on to, actually, one of the coolest topics that really kind of finishes this whole section of the class, and that's the Ellipsoid Method. So we'll look at this, and I'll try to make clear what is useful and what's not. The Analytic Center Cutting-Plane Method is useful. When you have a problem that you need to solve where you really do only have access to something like a cutting plane or sub-gradient oracle, and for whatever reason – you have to look at those reasons very carefully.

But if you have such a problem, this is an awfully good choice, and it's going to beat any sub-gradient type method very much. This is going to have much more computation, obviously, per step, more storage, all sorts of stuff like that, compared to sub-gradient method, which involves essentially zero computation and zero storage. They're way, way, way, way better than sub-gradient methods. Okay, so here's the Analytic Center Cutting-Plane Method. You're given an initial polyhedron known to contain some targets which might be feasible points, might be epsilon sub-optimal points.

Doesn't really matter what it is. You find the analytic center of the polyhedron, which is to say more precisely, you find the analytic center of the linear inequalities that represent the polyhedron. And you query the cutting-plane oracle at that point. If that point is in your target set, you quit. Otherwise, you add, you append this new cutting-plane to the set. Of course, what happens now is your point X (K+1) is not in P (K+1) by definition. Well, sorry. It might be in it if it's a neutral cut, but it's on the boundary.

What you need to do now is at the next step you'll need to calculate the analytic center of that new set, and I think – I won't go through this. There's a lot of different methods. Infeasible Sartinutin Method is the simplest one. Maybe not the best, but we'll go on and just go to a problem, and I'll show how this works. So this is a problem we've looked at already several times. It's piecewise linear minimization. It's a problem with 20 variables and 100 terms, and an optimal value around one. Let me add, just to make sure this is absolutely certain and clear. If this problem were – if you just had to solve a problem like that, it goes without saying you would not use something like an analytic center.

You'd just solve the LP. Let's bear in mind that every one of these iterations requires an effort that's at least on the order of magnitude of simply solving this problem to ten significant figures right off the bat by Barrier Method. So your code from last quarter, your homework code which shouldn't have been too many lines, will actually solve this problem very, very quickly. In fact, anyone want to take a cut at how fast it would be?

You have to solve – you have to minimize piecewise linear function 20 variables and 100 terms. It's gonna be an LP – when you add another variable, it's an LP with, I don't know, what is it, 21 variables and 100 constraints. Something like that. So let's say it's 20 variables, 100 constraints. Dominating cost is gonna be actually forming like A transpose

HA. So forming a 20 by 20 matrix, which is actually a matrix – 100 by 20 matrix multiply, or something like that. So that's 100 times 20 squared, and you're gonna do maybe 20 steps. That's an interior point method. How fast would that be, just an order of magnitude?

What do you think? Microsecond. Well, it's good. You're guessing sort of the right numbers now. You might be a little bit low. Actually, when you get down into the 10 and 20 variables, the N cubed, the various blast extrapolations start falling off. But let's – based on blast extrapolation – well, I can tell you how I'd do it. It's 20 squared times 100, and the way I do it is I think 1,000 cubed. Let's just say a second. That's way more 'cause a Cholesky factorization in this – I don't know. It's .18 seconds on my laptop, so it's not a big deal.

But something else matrix [inaudible] might be half a second. So let's call it a second. Let's round up and say 1,000 cubes is a second. You'd get a factor of ten right here because you're doing 100. You get ten, and then you get 50 squared, 2500. So it's 25,000 times faster than one second. Let's say that's 40 microseconds, and let's be generous and say you're gonna do 20 of those steps. I think I did that. So 40 microseconds [inaudible] 20s. So the answer is a millisecond, so if there wasn't a microsecond, but your answer was in the right spirit.

All this is just to point out that this particular problem could be solved easily through very high accuracy in under one millisecond. These are very good things to know, by the way. Very few people know this, I promise you. If you go around and ask people who – if you ask the authors of widely used packages, they have no idea how fast you can solve small problems.

Student: Is this leaving – every time you make a cut, is this leaving?

Instructor (Stephen Boyd):Yes. So in this example, this is gonna leap all constraint. So it must be understood here that when you start, we probably had 40 constraints because we probably put a box around it or something. I don't – I'm sure we did that. So we probably put a box around it. This has forty constraints, and in the classic ACCPM thing, how many constraints have you got over here? That's it. 240. So the number of – back to the size of the problem solved over here is six times as big as the problem here. It's 240. So we'll get to a bunch of this. We'll talk about it and just see how it works.

Nice thing you see here is you don't see that horrible slow-down you don't see in a sub-gradient method where you kinda go like this, and then kinda slow – get into this nice, one over K, very slow convergence. What you do is you see something that looks much more like an interior point method, except that would not be 200 steps; it'd be 20, or something like that. You'd see nice, geometric, linear conversions.

Okay, now the previous one shows – this is F of X, K – F*, and what this shows is the best of those. So this is probably what matters because when you terminate the algorithm at any point, you're actually – of course, what you're gonna return is the best thing you've seen so far. These long things here – of course, well actually, what do they mean? What does the long horizontal – what does a segment mean in this picture? This is iteration number versus sub-optimality. Your best one.

Student: Trying to satisfy the constraints.

Instructor (Stephen Boyd):No, this is – we keep all constraints. What does it mean?

Student: It's infeasible?

Instructor (Stephen Boyd):No, it can't be infeasible, right? Because we just – well, if it's infeasible, we stop. But that can't happen for this piecewise linear minimization problem. No, what this means here is that your F-K best has not changed. It means, in this case, that for ten steps you produced points that actually were worse than the best thing you've already seen. This is just underscoring the fact that this is a non-descent method. So a flat part means that some people might say that you failed to make progress, unless you failed to make progress in terms of finding a point with a better objective value. Actually, how have you made progress? What actually did happen in one of these little steps?

Student: You have a more constrained space.

Instructor (Stephen Boyd):Yeah, so your P-K got smaller. So, technically speaking, your ignorance was reduced over that step, but you failed to produce a point. You were lucky. You had a good point, but your ignorance – and that ignorance is going to translate into future good point values. So that's what those mean. Okay, now if you want to compare this to Newton's steps – and I think if we go back and forth, we can look at the two pictures. That's about 200 iterations, and here you can see that we did about, I don't know, about ten, eleven Newton steps. About eleven Newton steps per – so it took, on average, about eleven Newton steps to recalculate a new analytic center.

By the way, I should mention that we're – this is not tuned. We did have a fairly small tolerance for analytic centering. In other words we probably calculate the analytic center to land the squared less when you minus six or something like that. Needless to say, that means that the last three Newton steps we took in every one of our centering was a complete waste of time. Because remember, the goal here is not to calculate the analytic center. The goal is to get enough query point to – so these numbers, you could probably get it down to five Newton steps. I'm just saying if you actually wanted to do this.

So this is just sort of straight out of the box. No – well, all the code is online, so you can look at it as well as I can.

Student: For the infeasible Newton's, don't you have to wait until you get feasibility?

Instructor (Stephen Boyd):Yeah, and in fact, we don't have that here, but I might actually go back in the code and actually get some of the statistics. Like how many steps on average – the feasibility. Precisely right, that would be a really good thing to know. I may go back and do that. It'd be fun. You can, by the way, since the code is all there. Yeah, so that would be one question. And you can see, by the way, a little curvature means that as this thing progresses, the centering are getting a little bit harder. There's certain – you're on a pretty steep thing like this for a while, and then the slope here is half as – if you compare the beginning and the end, the centering have gotten harder.

By the way, what is the effort per Newton step? Is this a valid – does this track time well? Is that – I mean, that's Newton's step. Does it track time? Is that a valid – so, actually, how does time grow with these?

Student: Well, you're adding an extra constraint.

Instructor (Stephen Boyd):Absolutely.

Student: Every time, so yeah. I think it's –

Instructor (Stephen Boyd):So it's going linearly. The problem size. What's the competition complexity of the Newton step as a function of the number of inequalities? It's what? You guys have to get good at this. Here, you want to know the sloppy way to do it? Someone just walks up to you on the street and says, "Quick, quick, what's the complexity?" The big number times the small number squared because that's the answer for lease squares and any problem like that, right?

By the way, that's conditioned on the fact that you've done it right. So if in fact it's the big number squared times the small number then it's wrong. So that's my quick rule of thumb. Then later, when you can go catch your breath or whatever, you can go work out the Newton step, and there's 20 of these. But that's just a good working number. You know, if M is bigger than N, it's M-N squared. That's the answer for Lease squares, by the way. Lease squares, Lease norm, everything. So therefore, the cost per Newton step – what's happening is you have 20 variables.

That's a small dimension, and then you have the big dimension is the number of constraints. And it starts at 40 and ends at, I don't know, 240. What did we say? Goes up by factor of six, and roughly speaking, it's growing linearly. So as you go along here, that's a six times bigger problem in terms of big dimension. Therefore, if you do things right, it's a big dimension linear, quadratic in the small dimension. The cost of a Newton step from here to here actually goes up by a factor of six, linearly. So what you do is you'd do a quadratic distortion of this axis to give you the actual time. Make sense?

Because each step over here costs six times what a step over here costs. A step in the middle costs three times. So all I would do is take these numbers and spread them out quadratically. I'd have an affine expansion. Everybody cool on this? So that would be time. We didn't have it here, but it doesn't matter.

All right, so this just shows you again. This is the base line, nothing funny, no constraint dropping, no nothing, just basic ACCPM, right out of the box. So here what we have done is we have plotted two things. This is the true optimality. Now, of course, when you're running, you do not know this thing. We obtained this by using CVX to solve this in not one millisecond. Let's see what CVX had us interpret it overhead. It's gonna swamp everything. I guess by the time it gets down to STBT3, it's probably 30 milliseconds here.

Wait, super fast. We've got the actual solution, and then use that to judge our progress. This however – so this you do not know in real time. What you do know is that, this number. This is the best upper bound. By the way, the lower bound is also – does not go up monotonically, right? In an interior point method, right, what happens is that every step when you do a centering is you get a better point as your objective value goes down. And not only that, your dual value goes up. So you get the beautiful things where your best point and – you don't distinguish between your best point and your current point because the current point is always the best one. It's a descent method.

But you're lower bounds also go up, so you have no logic in an interior point barrier method, or whatever, that says keep a hold of the last point because it may be a while before you see one that's better. Or in terms of lower bounds, you don't keep track of the best lower bound you've seen because the next 15 you see are actually worse than the one you have. So here, though, you do know this one, and you can see it's off by some factor. I don't know, a factor of eight or something like that, or whatever. So that's all. That's all that this is.

So your stopping criterion might be based on the red dash thing, although I should probably say something. The sub-gradient method has, in general I think it's fair to say from a practical point of view, no stopping criterion. But usually when you're running sub-gradient methods, it's in some situation where you're so desperate to have anything that's all – that does anything, that you're just happy to have something. If things get better after 50 steps, or ten, you know. That's good enough, right? So stopping criterion is a luxury that you, generally speaking, don't actually have when you apply one of these.

The same might be true here in these. Maybe less so, but you still have the same thing. Okay, now we do constraint dropping. All right, so here what we did is – now by the way, keeping 3-N is actually quite interesting because we start with 40 constraints. We start with a box on X. X is in our 20. We put it in a box. I'm guessing that, but one of you with a laptop could actually look at the code and find out. You have a laptop. You don't have to, but I presume that's how we initialize it is with 40 constraints, which are BOTS constraints. So – and this is the progress if you do no constraint dropping the blue thing.

And remember, when you know dropping up here, your polyhedron has 240, or whatever it is. Wait a minute. Yeah, up here. It has 240 inequalities. So that's this thing at R-20, polyhedron at R-20. Kind of a small one, by the way, at this point. It's quite small, and it's got 240 inequalities. By the way, I have no idea how many of the 240 are redundant. I presume the original 40 are redundant. The original 40 were the boxes, and the idea is we just made a big old box. So it would kinda be bad if those were still active. So those 40 are probably redundant. Probably a lot of redundant, but I don't know. We didn't do that calculation.

But we keep 3-N. So we're going to keep 60 at any given time. So by the way, a polyhedron in R-20. What's the minimum number of inequalities that would give you a bounded polyhedron in R-20?

Student: Forty?

Instructor (Stephen Boyd):Forty, well that – in R-20. A 40 would do it, absolutely, if you have a box, but that's not the minimum number. What's the minimum number? Can you do it with eight?

Student: No.

Instructor (Stephen Boyd):What? Someone said no. Why not?

Student: It's 20 –

Instructor (Stephen Boyd):So you need at least 20. If you have eight, then basically there is a 12 dimensional sub-space of things orthogonal to those. And so you're unbounded this instantly in at least 12 dimensions, so you're not gonna be bounded. So the answer is 21. You need 21, and a simplex is the smallest number of – it's the simplest polyhedron that's bounded. Simplest in the number of inequalities, and it requires N plus one.

So the point is it takes 21 inequalities. To just have a bounded polyhedron, we're taking like 60, so it's not like it's a super – this is not an exquisitely detailed description of a polyhedron. And the wild thing is you see this? It's just totally insane. The progress is identical. By the way, what is the change in computation time? Again, not that we would care, but lets figure that out. What's the difference between solving – computing the analytic center for the red guy and the blue. What is it there?

Student: Post for each Newtons.

Instructor (Stephen Boyd):Yeah, I know that, but I want to know the number. You can figure it out. Everything is here. You know all the dimensions, so I want the number. Of course the cost per Newton step is different. That's what it is. Now I want the number.

Student: When you drop, or when you don't drop?

Instructor (Stephen Boyd):Both. The blue is no drop. The red is keep 60, so you're dropping. What is it?

Student:B minus 6 –

Instructor (Stephen Boyd):Do 40 over 60. So the – in this case, you analytic center 240 inequalities, 20 variables. In the red case, you analytic center 20 variables, 60 inequalities. So, again, big times small squared. It's linear and big. 240 versus 64 to one.

Student: What about the complexity of reducing the constraints?

Instructor (Stephen Boyd):Oh, that would solving like an LP per step. Oh, now we could actually do much more analysis in here that would sort of be fun. So you know an Analytic Center Cutting-Plane Method. At each step there is – you get for free – some constraints you can promise. You can actually guarantee are redundant with no further discussion. You don't have to avoid your complexity theorist friends or anything like that. Everything is cool because they're provably redundant. Actually, I would love to have figured out how many – at each step we drop things, obviously. Well, we actually didn't drop things until here, right?

Because we started with 40, and then 41, 42, 43. When we got to 60, we started dropping, and then it looked like that, right? It would be a plot of the number of inequalities. So we didn't have to look the other way for the first 20 steps. After that, when we start dropping, the question is what fraction of the things we dropped were provably or actually redundant. All of those could be calculated by just solving some LPs. Those would be really good numbers to have. We should have them in here. It'd be fun.

It is very unlikely that everything we dropped – oh, would that be right? Oh, hang on. Oh, that's not true, sorry. I was about to say something that was wrong. Good thing I didn't say it. After all this discussion, though, let me make sure that the big picture points are clear. So the big picture points are the following. It obviously works just as well. You don't need the number – there's no particular advantage in convergence you are getting by keeping all these constraints. That by limiting it to 3-N, you're getting the full power. You're getting perfectly good convergence.

The computational complexity like that means that the cost per set here is actually – in one case it's growing, and in another case it's just flat fixed. So that's the main point. By the way, proofs of convergence that handle constraint dropping are quite complicated, as you might imagine. They're quite complicated. So they exist. I'm very happy somebody has done it so that I can tell you somebody has done it. There's a gap in between, I might add. I'm pretty sure there's a gap in between the things people do, and the things people have proof converged.

So I do not know if a method that were – that keeping 3-N is actually convergence. I actually don't know that. Maybe it is, maybe it isn't. But I've seen ones where you drop things, and these are quite complicated, as you might imagine. Okay. Oh, I guess we figured this out on our own already, so there's nothing to say about this. The sort of numbers steps, and the number of inequalities, it's kind of a silly plot, right? But, anyway, it's – okay. So now we get to something interesting.

This is using a mega flop counter in Mad Lab which is notoriously bad and all that, and I think people don't even use it anymore, but it doesn't matter. Just to get a rough idea, we should be able to figure out – actually, we should be able to guess a lot of these things really, really well. So let's guess. Actually, I'll tell you how we can guess this plot. Let's go to the end, and ask what is the accumulated – we already decided that the red thing at this point is six times more efficient than the blue.

Student: Four.

Instructor (Stephen Boyd):Four, okay, sure, whatever. Yeah, four. Okay, so it's four times – and how much faster is it in the middle. Oh, I don't know. It's 60 versus 40 plus 100, 140? Close to two. This is a street fighting method. This is quick, dirty methods, fast. If it's four times more efficient and it's two here, what's sort of the average inefficient? And it kind of grows linearly or something. What's the average sort of advantage? Two, I don't know. What do you think? Well, it's not eight, okay? And it's not four, and it's not one.

Let's call it two-to-one, and let's see if our prediction is pretty good. Our prediction is, oh, hey. What do you know? Think about 150 mega flops, and this took about 70. So that's the picture. So this shows that constraint dropping, if you care about time or whatever, it works this way. By the way, a lot of this doesn't matter so much because you usually use analytic center cutting-plane in a situation where the oracle calls. The sub-gradient calls are very expensive. For example, the oracle calls – in many of the most interesting cases, this is when you're doing distributed optimization.

These are like other computers on the other side of the world. All sorts of stuff is happening. So the analytic center cutting-plane stuff is just like zero. Even for a problem with 10,000 variables, or something. That's how these things work. Okay, now if you do the epigraph analytic center cutting-plane method, now you have 20 variables and 100 terms. And what happens is you divide – so this actually goes way down here.

So you actually are dividing the number of steps required by just a factor of four, or something. And that's it. I don't think I'm showing F best here, am I?

Yeah, there we go. So this is the epigraph one, and here's what it looks like versus Newton's steps, and I don't know what happens here. But you – I mean, it looks like you're doing – there's points here where your doing better, or something. And the word on the streets as I know is that you should – is that the Epigraph Method is better. I don't understand it, to be honest. I'm not quite sure why, but anyway. For whatever it's worth. Okay, so this finishes up analytic center cutting-plane method.

By the way, I should say that we haven't really seen yet the really cool examples where you would want to use these methods. So – and I admit that. That's going to be the next chunk of the class, and we'll see some very, very cool applications where you really would want to use these methods. In fact, you will. So the next topic is just absolutely beautiful. You can guess where it's from.

Student: Moscow.

Instructor (Stephen Boyd):You're right, actually Moscow and Kiev. So it's from Uddin Nemerofsky Shore in Kiev, and so and so. Guess when. Close. Actually, it probably is 60s. They say definitely for sure early 70s, and yeah. I'm sure they knew it in Moscow in the 70s, or some 60s. And it's absolutely beautiful. By the way, it's quite famous, too; not very well known in the West. By the late 70s it was on the cover of the New York Times front page because it was one of the first papers used to show that linear programs could be solved in polynomial time. By the way, with a little star, meaning you have to go read the fine print.

But that was 1979, and they had a hilarious article on the front page. It was really funny, actually, and it said "Linear programming runs the entire world, everything. All airlines are scheduled for it. Nuclear weapons are built by it." It just went on and on and on, and said this is going to totally change your life because now, and by the way, I should add. At that point for 30 years, people had been solving linear programs unbelievably well using a simplex method, just so you know. Amazingly well. It was a non-issue.

They called some mathematician at Rutgers. I mean, actually, the poor guy wrote the article. All he said was there was a complexity theory result, and it was a pretty good one. By the way, up until that point, people didn't know what the complexity of solving linear program was, and they actually introduced a new class called LP, as in P. So you'd actually say this is class LP, and it was as easy to solve as an LP and vice versa. If you had a method for solving an LP, you could actually solve these things. You had to reduct the two-way reduction. So, yeah, they really did, and then it turned out P was LP or something.

What I'm saying is that if you really look into these things you'll see it's much more complicated and all that, and some of the things I'm saying, if interpreted literally, are just wrong. So this is on the front page of the New York Times saying it was gonna change the world, and it didn't. Linear programming was on the front page again in 1984 for barrier methods. This is just absolutely beautiful, this method. This is the one I just talked about. This is Gocheon, who by this time was at Rutgers, used this to show polynomial salability of LPs. So, actually, you're gonna see a code now.

What's funny about this is that the sub-gradient method is to solve an LP. I think that maybe you wrote a – I mean, it's not hard to write a six line code for it, and a sub-gradient method. You can just write a sub-gradient method that solves any LP. Even more, if someone says, "That's ridiculous. It's hard to write an LP solver. These are big, giant pieces of software you have to know a lot." The proof is like a paragraph because it's the paragraph from sub-gradient method. That's pretty cool. Slower than anything, right? But it is nevertheless cool that a six-line program with a two paragraph proof solves any LP.

This was gonna be eight lines, but even scarier according to the complexity theorists this is actually an efficient method. So – and the proof is quite short, which is even more shocking. Each step is going to require cutting-plane or sub-gradient evaluation, so it's the same as everything else. You're not gonna have mono-storage, which is to say it's gonna be Ovan squared. Which by the way is, if you do analytic center cutting-plane method with dropping, the storage is N squared because you have N variables and you store three N, five N constraints. That's all you need. The only state in that algorithm is the polyhedron. So your storage then is like, whatever. Six N squared. That's it.

And your update cost, by the way, is N cubed in that one, obviously because it's N squared. Whatever it is, it's the big times the small squared. You can actually make it faster with rank updates, but that's another story. Okay, so it's got mono-storage. In the Ellipsoid Method the update will involve solving no – well, you will solve a convex problem, but it has an analytic solution, so – and it's gonna be order N squared. And sufficient in theory, and it's slow in practice, but it's actually something like the sub-gradient method, actually, except that it's fed in the sub-gradient method. It's kind of slow in practice, but it is way robust.

You can blow off whole parts of it, and start it in the wrong place, initialize it wrong, and you can give it totally wrong sub-gradients many, many times. It'll just happily lumber along, and work really well. So that's – okay. So the motivation goes something like this. In a cutting-plane method, you need serious computation. I don't know. Do you really believe that? It's ten Newton steps. Each Newton step is a Lease squares problem. Okay, so it's ten lines, but this scares people, and people make a big deal about an analytical formula. You know, like a Kalman filter update versus if there is a conditional or align surge, they get all weird about it.

Of course, that's completely stupid and idiotic if you think about it right because a Newton step is nothing but a Lease squares. So I don't know why people – I mean, it's fine to have an analytical formula, and that's all great and everything, but the idea that these are sharply divided is really, really dumb. It's a distinction that makes absolutely no sense. What only matters is how fast can you carry out that calculation, and what is the variance in that calculation. And nothing else matters.

The variance can range from zero if you're doing a matrix multiply to a little tiny bit of – if you're doing a sparse matrix factorization, considerable. I mean, it depends on the – with a not given structure you have huge variance because it depends on who did the ordering for you, and how much filling you get and all that kind of stuff. Do an Eigen Value calculation for dense matrix. It has a very small – it's order N cubed, but it has an N squared component – or an order N component that is subject to some variation, which is the number of steps required to calculate the Eigen Value. Obviously, if it's N cubed plus a random number times N, the way to summarize that is it has no variance.

Oh, we just talked about this at great length. The other issue is this, is that the localization polyhedron grows in complexity as the algorithm progresses. We already talked about that. Now, I have to say, if you keep it proportional to N, like this – so this second bullet is not a problem in practice. If you do analytic center cutting-plane method, you are going to prune constraints. Okay? That's the way it's – that's how they're done in practice. That's how they work. They appear to work juts fine, in which case this is not an issue.

So this only irritates your aesthetic side, or your theoretical side or something like that. So that's all that is. The Ellipsoid Method is gonna address both. It's gonna have a constant data structure to summarize your ignorance at each step. Analytic center cutting-plane method has a variable size because it's a polyhedron with a variable number of constraints, and in the no cutting method it just grows.

So here's the Ellipsoid Method. The idea is that you're going to localize the point you're looking for in the ellipsoid instead of the polyhedron. By the way, that's actually kind of a very cool idea right there, and ellipsoid actually is – what's required to describe an ellipsoid? What's the size of a data structure in RN? What do you have to describe an ellipsoid? There's lots of different ways to do it? What do you need?

Student: Matrix and a vector?

Instructor (Stephen Boyd):Matrix and a vector. And a matrix is N squared over two, and a vector is like N, so – in fact the matrix is like N and plus one over two. It's N squared over two. That's the answer. So the number – to describe an ellipsoid, the data structure for an ellipsoid requires about N squared over two elements. Oh, let's see. Let me ask a couple more questions. Oh, and an ellipsoid is an interesting set. They're very widely used in RN, for example, if you do statistics or anything like that, it's sort of the first thing that gives you some direction. So let's just talk about localization sets just for a minute.

Here's one: bounding box. So you have a box. Someone says to you during a calculation, "Are you doing navigation?" and you say, "Okay." And someone says, "How accurate are you?" You'd say, "Well, give me plus minus two meters east west plus minus north south in elevation, such and such. Plus minus eight meters or something."

That's a bounding box. Widely used. How many – how big is the data structure required to describe a bounding box? An RN. It's two N. It's an L vector and a U vector. That's it. So that's two N. Okay, so that's – these are at the simple methods, right? Here's an even simpler one. How about just a ball? What do you need to describe a ball?

Student:N plus one.

Instructor (Stephen Boyd):N plus one. You have to give the center and a radius, so it's N plus one. So these are order N data structures that describe geometric sets, and they have uses. They have plenty of uses. Actually, for most people this is the simple – they shouldn't be made fun of. They're extremely useful when somebody says, "How are you estimating the velocity of that vehicle?" and you can say, "Plus minus .1 meters per second."

Everybody understands that. It makes total sense. An ellipsoid is already the first sophisticated thing. You see it in statistics because you talk about confidence ellipsoids, and what it captures is in statistics correlations. So it allows you to capture the idea that you could know X plus minus a meter, Y plus minus a meter, but for some weird reason, you know, X plus Y to ten centimeters. That's this kind of ellipsoid, right? So that's what an ellipsoid does. That's what an ellipsoid gives you. So you're into the N squared things.

How about a simplex just for fun? Just roughly what's the data structure? Think of the simplex. Not a unit simplex, a general simplex. Convex hold and plus one points. I just gave it away. It's N squared, roughly. Then it turns out you have to adjust for that because they're symmetric. It's order N squared. It's actually a big jump when you go from an order N to an order N squared data structure described geometric set. You actually go from not the inability to universally approximate to ability to universally approximate.

Universally approximate means you can – I can give you any old convex set, and you give the inner approximation of whatever data structure you choose that has some bound that says that if give an outer one, if you shrink it, it lies inside. Ellipsoids do – bounding boxes do not do that because you make a little razor thin thing that goes at a 45 degree angle, and you're in big trouble because that bounding box is huge, and you have to shrink it infinitely spar before it fits inside. But an ellipsoid, you have a bound. We did it last – or we saw it squared in, or whatever. N is the factor for an ellipsoid. Squared N of the symmetric. Okay, lets go on, it's just an aside.

So here's what's going to happen. You know – the idea is that you have an ellipsoid to summarize your ignorance. You know the optimum point is a current ellipsoid. Now the nice thing about an ellipsoid is there's no long discussion about centers. It's sort of like an interval, an ellipsoid. So you can say, "Well, let's calculate the analytic center of an ellipsoid. Let's calculate the CG. Let's calculate whatever you want."

They're all the same. In fact, if you have a center of an ellipsoid and it's not the center, then I'm deeply suspicious of your definition of center. So the point is there is no real discussion there. There's just the center of an ellipsoid, and you evaluate a sub-gradient at that point. And what you do – that tells you instantly of a cutting-plane. So what happens is you know that your optimum point – you're in an ellipsoid. You have sliced off a half-plane, and you have a half ellipsoid. And now you know – now, if you did this a standard cutting-plane method, you'd calculate now – by the way, you have a half ellipsoid, and you start about the center. Now the center is varied.

There's an analytic center. There's a CG. Then you do it two steps, and you have now a set, which is an ellipsoid sliced by two planes, and that's – so in fact, there's another step that goes like this, and this is the main point. Here's what you do. You have a half ellipsoid, and in order to preserve constancy of the data structure, you're now going to do something that's equivalent to constraint dropping or whatever. You're gonna set E-K plus one be the minimum volume ellipsoid that covers your half ellipsoid. Okay?

So step four – step three is where you get your knowledge, your ignorance decrease. So this is ignorance decrease, and in fact, by the way, how much does the volume go down between E-K and this localization set?

Student: Factor two N?

Instructor (Stephen Boyd):Yeah, a factor of two. If it's a neutral cut, if it's a deep cut, it's more than a factor of two. So in step three you get at least one bit. In other words, log two of the volume of the localization set, and step three goes down by at least one. The worst thing that could happen is your ellipsoid is cut in half. No matter how you cut an ellipsoid, every plane going through the center of an ellipsoid cuts the volume exactly in half. Your volume goes down exactly by half here. Now, in step four your volume goes up. So four is actually an ignorance increasing step, and then there's just a big drum roll to find out whether or not the ignorance increase in step four is more or less than the ignorance decrease in step three.

That's basically the method. Yeah?

Student: How are we finding that minimum volume ellipsoid? Are we keeping track of half-planes?

Instructor (Stephen Boyd):No, I'm gonna show you. There's gonna be just a formula for it, yeah, worked out by some Russians. The basic one is simple, but they did things like two cuts, and they actually had formulas for deep cuts, and parallel cuts, and crazy things. And then you get into some analytical formulas for those. Those are formulas that believe only a Russian person would work out, I believe. But there are formulas for it. But this one is gonna be simple, and I think you're gonna work it out because we're gonna stick it on the homework. Do we have an exercise on that? We just calculate the minimum volume ellipsoid that covers a half ellipsoid.

Student: We have one on that.

Instructor (Stephen Boyd):Oh, we do? Okay, but not yet assigned?

Student: No.

Instructor (Stephen Boyd):How convenient. What do you know? So here's the picture. Picture goes like this. Here is your ignorance at the K step. Now, by the way, some points in this thing could not be optimal because remember, you're cycling between steps that decrease ignorance and then increase it. So first you decrease ignorance. You call, you get a – this is a neutral cut, like this, and basically, this half ellipsoid, we do not have to look in ever again. So actually, at this half step, we can say that the solution lies in this half ellipsoid. Okay? Our ignorance just went down by exactly one bit.

Now the next step is we calculate the minimum volume ellipsoid that covers it. We're gonna get an analytic formula for that. Here it is. It's this thing. And what's cool about it is things like this and this are very interesting. That’s just slop. We have now – before this step, we know the solution cannot be in this little thing here. It cannot be there, and yet, we’re going to include it in our localization step as we move forward. Okay? So that’s the bad part. That’s the slop.

Now compared to a cutting-plane method, you can state various things like localization set doesn’t grow more complicated, it’s easy to compute a query point – well, yeah because it’s actually part of the data structure of the ellipsoid, right? An ellipsoid is given by a point, the center, and some positive, definite matrix. Okay, fine. It’s an access into a field of a data structure. The bad part is, of course, we’re adding unnecessary points in step four. Anyway, I wouldn’t be here telling you about this if it didn’t actually work. We’ll get to that.

So the first thing is ellipsoid method in R reduces it by section. Good, very good. So R, what’s an ellipsoid in R? It’s an interval. And then it says go to the center of that interval, and get a sub-gradient. It basically tells you the left or the right is your new localization center, and indeed you have gone down by one bit exactly. And now you have an interval, and you call a method that says please find for me the smallest length interval that covers this interval. So I can write that method, that’s easy. In that case there is no slop. But this example back I just showed the visual example shows immediately. In R-2, they’re slop. You are adding points into your localization set that you know for sure cannot be optimal.

Okay, so we’ll get to the formula – the update formula. That’s easy. Now it turns out that this – the new localization set can actually be larger than this in diameter, but it’s always smaller in volume. This is how the drum roll ends, and it’s in fact more than that. It turns out that the volume of the new ellipsoid is less or equal to E to the minus one over two N time the volume of this one. Now, the good news about this number is it’s less than one. That’s the good news. The bad news is that it’s very close to one. Shockingly, that is enough to give you a polynomial time result, which we’re just gonna do in a few minutes.

So does that make sense? So compare this volume reduction factor. In CG, what number goes here? Oh, I already gave it away, it’s a number. Sorry. So what expression do you put in CG here?

Student:163?

Instructor (Stephen Boyd):163, right, which does not degrade, which first of all is a very respectable number. It is not .999. That’s number one. Number two: It does not degrade. The CG method does have a few problems. For example, the sub-routine that you have to call to calculate the CG is far harder than solving the original problem. But still, you can see what happened. So this is a very slim amount. MVE you get something that is much better. I forget. It’s something like one minus one over N. Is that right? It’s in the notes. You just have to look back on a lecture, but let’s say it’s something like that.

It degrades with MVE, but nothing like an exponential. So let's look at an example and see how it works. So you can't see it, but there's some sub-level sets of convex function here. It's not exactly polyhedral, so I don't know what it is. I probably logged some X for something like that. So you start here, and sub-gradient – probably gradient in this case points that direction. And that says that this half circle cannot contain the optimum. I think the optimum is somewhere around here. Okay?

That's your localization set, intermediate. This is half circle, and you take that half circle, and you cover it with the smallest volume ellipsoid that covers it, and that's this thing. Then you move over here. That's this one. You go to the center. You call sub-gradient. You get this. This half ellipsoid has now been ruled out, and you have this half ellipsoid, and then you jump over here. And I think just looking at these, since this is obviously – I reject the idea that there is, I mean – there is no problem minimizing convex function in R-2 because you just grid things. That's a non-problem.

But still, the fact where you have a non-problem, you can sort of visually see that it's going to be slow. It's sort of a hint on these things, and I won't trace through these, but that's the next three steps. You kind of get the idea. So now let's talk about how you update an ellipsoid to cover half an ellipsoid. By the way, I'm doing – this is the formula for a neutral cut. So how do you cover a half ellipsoid with a minimum volume ellipsoid?

We also do deep cuts, and in deep cuts you have to ask how do you cover a cap? Not a half ellipsoid, but something where you cut a fraction off by a minimum volume ellipsoid. There's a formula for that. It's not too bad. But let's look at this one where you update the ellipsoid this way, and I can – let me just tell you why. This is going to be on your homework anyway, so I'll just say a little bit about why it's not that big a deal. Let me ask you this. I can change affine change of coordinates, and this problem is the same.

If I create an affine change of coordinates for an ellipsoid – when I do an affine change of coordinates, all volumes get multiplied by the determinant – the absolute value of the determinant of the transformation matrix, right? Therefore minimum volume – I can transform coordinates, minimize the volume, transform back, and I got the answer because everything just got multiplied. So basically, I don't have to consider this a general thing. I can transform any half ellipsoid I like to half of the unit's sphere, half of the unit ball just pointing upwards, or whatever X one bigger than, or X-N bigger than zero.

So that makes it simple. Now let's just do this visually, and you have the homework problem. If you have half an ellipsoid, it's completely symmetric in the remaining and minus one dimensions. In one dimension it's got to be positive, but the other one. So that says that the ellipsoid you cover it with has got to actually be symmetric in those dimensions as well. That's a standard thing we've seen in where convex authorization. If the problem has symmetry, the solution must in convex authorization. It has to have the same symmetry.

By the way, that is absolutely false for non-convex problems. So, for convex problems, any symmetry in the problem, you can immediately assume that the solution will also be symmetric under that group of symmetries. Okay? So in this case, since you have a half ball that is sticking a cap like this, sticks like that. Then it's symmetric in all of these other dimensions, and therefore that means that the P matrix is down to one variable or something. I think you have two variables to mess with. By the time you're down to two variables, it's calculus time. It's not a big deal. It's what you learn calculus for. It's the only thing you learn calculus for.

So here is the update. Oh, for N equals one, we do know the minimum volume ellipsoid that covers a half ellipsoid. That's easy. That's basically itself because an ellipsoid is an interval. A half ellipsoid is another interval, and I know the minimum length that covers an interval. So, okay, the minimum volume ellipsoid looks like this. Here's the update. Let me just say a little bit about what the data structure is that we're gonna use to describe. Lot's of ways to describe ellipsoids, right? Lot's of ways, or at least I know three or four. Image of uniball, inverse image of a uniball, quadratic function, and lots of things.

But we'll do it this way. We're going to parameterize it by the center, and a matrix P that's positive definite. In this formulation here, things like the volume is proportional to something like [inaudible] to the, I guess, square root, or something. So the bigger P is – I guess the point is that roughly speaking, the bigger the P is, the bigger the ellipsoid. Did I get that right? Yeah because if P is big, it basically says P-N versus small, and you can go Z minus X can get very big before this thing runs up against a limit of one.

Student: What part of P is vague?

Instructor (Stephen Boyd):Well, yeah. That's vague. What does it mean so – I was being vague. When you say a matrix, a positive of a matrix – P is big. When you pin somebody down, it can mean lots of things. It can mean big in determinant, big in diameter, big in the smallest Eigen Value, so I just mean hand waving. What I'm doing. You can talk about which directions it's big in, right? You can talk about the Eigen. So the Eigen values of P tell you – for example, the condition number P tells you how non-isotropic or an-isotropic that ellipsoid is. It tells you if the condition number is like two. It says in one condition it's twice as big as it is in some other. If it's 10,000, it says something else and so on. All right.

So let's look at it. Here's your – the new ellipsoid looks like this, and it's really cool. I'll give you another interpretation of this in a minute. So the new ellipsoid looks like this. The center – of course, it's very interesting. It's a step in the direction P-G. I'm gonna talk about that. You step not in the direction G – negative G. But in the direction P-G, so in a sort of change of coordinates you step in a different direction. That's this. Step length is one over N plus one, and the new ellipsoid looks like this. And I want to dissect the different parts. Remember, P big in matrix sense roughly corresponds to E big.

So lets talk about the different parts. Let's forget the scaling, and let's focus on this. This – by the way, what is this matrix here? What's the rank of that guy?

Student: One?

Instructor (Stephen Boyd):One, rank one. It's an outer product. It's like P-G times the transpose, okay, with a constant in front. This is actually – that's a rank one down date. In other words, you take a positive definite matrix, and you subtract a rank one thing. I mean, you can't subtract – this thing is carefully – well, by definition this cannot be positive. This is, again, positive definite. You know what that means? If I ask you about – suppose let's take the ellipsoid defined by P, and then the ellipsoid defined by that. What can you say about the second ellipsoid? What can you say about all the Eigen values of that matrix compared to the Eigen values of that matrix? You can guess.

Student: They must have gotten smaller.

Instructor (Stephen Boyd):That's true. They all got smaller. This matrix is less than or equal to P in the matrix sense. It got – it shrunk. Actually, it only shrunk in one direction. It actually shrunk in the direction P-G, or whatever direction this is. It only shrunk in one direction, okay? But it got smaller, therefore this is – that's the good part. This is the good part where P gets smaller. Then you multiply it by this, and what happens to the ellipsoid? Now what happens to the ellipsoid when this happens? You increase it. So basically, I don't know if people have seen this in Kalman filtering, but basically what happens is – again, this is only if you've had these classes. If you haven't, then don't worry about this.

You'd have two steps, right? What happens is first you take a measurement. So you take a measurement, and based on the measurement your estimate of the current state gets better. So your co-variance matrix goes down. It had to. You just got some information, and if you know how to process the information correctly, how could your estimate get worse? So your area co-variance matrix goes down, okay? Then the next that happens in sort of like a Kalman filter is you multiply. You propagate it forward by the dynamics, and add some unknown process noise, and what happens then is that in that step – that's the dynamics update. Your ignorance increases.

And the hope is when you're running this whole thing after many steps is that the amount of increases in ignorance – in that case measured by a co-variance matrix, is overwhelmed by the amount of decreases in ignorance that come every time you take a measurement. So that's what's happening. You have a vehicle or something you're tracking. It's being disturbed by something you don't know. That's a process that's leading to increased ignorance at each step, but you get noisy measurements of it, and that's a process which if you process those measurements correctly can lead to decreasing ignorance.

This is the same thing. That's the decrease, and the decrease came from kind of a measurement. If you want to call this a measurement, well you can. That's a measurement, and then that's your slop, so that's how this works. I can actually – let me give another interpretation of this. Anyway, you're gonna prove this formula, so here's what I'm gonna do. I'm gonna write down an interpretation of that, and once you see this interpretation, you don't need to know anything else. So it's this. Let me see if I can – I just want to interpret the step.

So this is how it goes. You have an ellipsoid like this, and here's your point. And you get a sub-gradient, and let's say the sub-gradient points this direction, okay? That's the sub-gradient. Now suppose you were doing a sub-gradient method. Where would you step in the sub-gradient method? You'd step somewhere along here. Now where depends on your step length on that step, and your step length can be as stupid as one over K, right? So you'd step in this direction. Let's actually see where you step here. Where do you step here? That's the question. I'll just draw it geometrically.

It turns out what you do is you go in the direction – you solve, actually, a problem. This direction that you step in solves this problem. Here's how you get the direction. You minimize G transpose V, where V is a direction subject to V in your ellipsoid. Okay? So you go as far as you can in this direction in this ellipsoid, and in that case – I'm going to try to draw it, and I'll probably get it all wrong, but anyway. It looks something like that, and so it might be here.

So in fact this is the direction – if that's G, that's minus P-G. You know, roughly. I mean, forget the scale factor. This guy down here is minus G, and so what you see is –you can think of it as a distorted – it's like Newton's method. It's exactly like Newton's method. Instead of going in a negative sub-gradient, you're twisting it by a positive definite matrix. Okay? This make sense? And you can check that this is – that P-G is sort of this direction here. And there's a beautiful way to interpret it, and it's this. And it's interpreted in change of co-ordinance.

So change of co-ordinance interpretation goes like this: Here is your ellipsoid at step N, or K, or whatever they call it. It doesn't matter. Here is your ellipsoid at some step, and what this says is – you know, the ellipsoid tells you your ignorance. If it's round, it says you're equally ignorant in all directions, right? If it's little pencil like things then it means that you have very good knowledge of the solution in some directions, and very poor in others, right? So what you do is you say no, and you change co-ordinance. And by the way, you change co-ordinance in the original thing by multiplying by P to the half.

So you use co-ordinate Z equals P to the half times X. You change co-ordinates to make your ignorance round. To make it isotropic so you're equally ignorant in all directions. You change co-ordinates. When you change co-ordinates, you get a transformed G. Let's call that G, I don't know, G – G had been pointing some direction. Well, it doesn't matter. This is your new – I 'm gonna call it G bar. That's G, but transformed by this P to the one-half, or whatever. Okay? Now in this case, the question would be which way to step?

And so the correct step in a situation where your ignorance is isotropic – so you're equally ignorant in all directions. The correct step is to go minus G bar – and it's even funnier than that. It's basically divided by N plus one. This constant step like whatever N plus one. That's it. That's what the ellipsoid method is. The original discussion of ellipsoid method, which was, I guess, by Shore, had this name. In fact, it had a very long name I'm trying to remember because it's really funny when translated into English. It's sub-gradient method with space dilation in the direction of the gradient.

So this is space dilation because when you're changing co-ordinates – and then, by the way, the dilation is only in one direction in each step because when you do just a rank one update, you're actually just scrunching space. Actually, you're expanding it. Well, you shrink the ellipsoid, and that has equivalent in the change of co-ordinates of accentuate of dilating space in the direction of the sub-gradient.

So when you look, if you were to go to Google or to look at some of these early Russian books from the '60s and '70s – some of which are actually in English and are superb books, you'd have a – instead of ellipsoid method, you'd hear ellipsoid method with space dilation in the direction of the – and various other things translated like that. This make sense? The thing you don't do in the sub-gradient method that you do do in ellipsoid is once you step this way, this ellipsoid gets shrunk.

You learn about stuff in this direction so you shrink a little bit, and you re-pull it apart. You do a change of co-ordinates to re-balance or isotropize your ignorance. That's definitely not a word in any language. Make it isotropic. This make sense? So this is the picture. So in some senses what's funny about this is it's a little bit like Newton's method. Because if someone says, "What's Newton's method?" and you can write down some formulas and stuff, and they, "Well, why would anyone want to do that?" and you can blabber on and on.

But if someone says, "Yeah, but what's the idea of Newton's method?" you can say that idea is that the gradient method, although it's the first thing you could think of – it's greedy, and it might look like a good idea to go downhill the fastest way that way, but it might turn out that you should really be going about 45 degrees over that way, or more commonly, 89 degrees that way. And someone saying, "Why?" and that's because you draw a valley and all that. And then you say that Newton's method is basically the gradient method applied after a change of co-ordinates makes the curvature locally isotropic. That's what it is.

So the one case when greedy is good is when your ignorance or function has kind of an equal curvature in all directions, and this is a sub-gradient method with this. In fact, people have a beautiful term for these things. It's called variable metric. The metric tells you about the topology of the space, so you would talk about how Newton's method is a variable metric, gradient method. And in fact, you'd talk about [inaudible] as a variable, metric, sub-gradient method. It has this other interpretation, and so on.

The stopping criterion is pretty simple, and the way that the stopping criterion – let me just draw that over here, and then let me tell you something embarrassing about the stopping criterion. So here is your current ignorance level, and suppose I give you a sub-gradient like that. Well, if you were gonna do another step, you'd cover this with an ellipsoid, and that would be your next point. And actually what you'd do – now you know geometrically what you'd do. You'd solve the LP of going as far as you can in that direction here, and in fact that would give you to this point, right?

So in fact your next step direction, now you know how to construct it, is here. And then the step length is somewhere along here, given by some formula. But you're not gonna do that. You're gonna terminate, and what you do is you stop, you send a signal to this process, and you say, "Done. Time is over. Give me your current point." And they'd say, "Please give me a lower bound on how far you are from optimal."

Well, what you do is this. You – of course you have this. This is just your basic formula for – everything is based on this formula. It's the definition of a sub-gradient. Everything is based on that. So what you do is this. X star is in this ellipsoid. The ellipsoid is a localization region. Therefore, this thing couldn't be – has to be bigger than or equal to the infimum of this linear affine function here over the ellipsoid, like that. It just has to be. You can just analytically work this out. I mean, what the worst thing is. In fact, the worst thing is – in fact, literally, what you're doing is you're solving this formula here.

You're calculating this point, and then when you put that back in – this just has an analytical formula. It's this. So it's absolutely beautiful. Square root of G transpose P-G is actually a guaranteed sub-optimality bound. Couldn't – and it's absolutely beautiful. Do you know what it is in this transformed space? It's this. It's the norm of the sub-gradient in the current co-ordinate system if you change co-ordinates. It kind of just makes beautiful sense, and it's a guaranteed sub-optimality. It makes sense because if someone walks up to you on the street, and you say, "Listen, consider a unit ball." That's kind of your standard ignorance set.

You say, "I have a unit ball. I evaluated the sub-gradient at the origin of a function, and I got this. I know the minimum is somewhere in the unit ball. How far could I be off?" You get this. It's the norm of G. That's how far you are off. Okay, so it's a beautiful stopping criterion, and especially because I think you calculate that as a side calculation anywhere. So you have to calculate this anyway. You calculate it right here. So basically, you calculate this. If it's less than a threshold, you're out. Otherwise, you update.

Okay, and I do have to tell you one thing about this. Well, here's the ellipsoid algorithm. I'll show you the whole algorithm. I have to admit something about it. So you start with an ellipsoid containing X star, you know, typically in the same way an ACCPM might be some box or something. You know, this might typically be a ball of radius capital R. That'd be very common. Okay, here's what you do. You evaluate a gradient. If the gradient in the scaled norm is less than epsilon, you return X – and that's actually certified. There's nothing fuzzy about that at all.

Otherwise you update the ellipsoid like this, and you repeat. That's the ellipsoid method, and now I have to tell you the truth. No, it's not the truth. We're not lying. It is not know, as far as I know, and I spent a week last summer with a bunch of people from Moscow and Kiev in the '60s. They were all there. Enough of them spoke English, so everything was cool. There was a lot of vodka, so that was on the other side, but it's not clear. Anyway, they claim the following: There is no proof that the exit criterion to will actually be satisfied.

So there's no reason to believe it. So the only actual stopping criterion is the one that comes from the complexity theory. Actual means that you can absolutely prove that it will be reached. However, this works invariably in practice. I think we already talked about this, so we already – okay. So here's an example of ellipsoid method. This is our now famous ellipsoid method, and you can kind of look and see how it's working. Each step is ordered in N squared, or something like that. Let's go back and see what the ellipsoid step is. Yeah, it looks to me like it's ordered N squared. Everybody else seeing N squared here for the ellipsoid?

The numerical update, I think, is N squared. The reason it's N squared is – I don't know. Here, you have to write all the entries of P out, so there's N squared right there. And other than that, I don't see anything fancy except I'm seeing [inaudible] vector multiplies. So it's order N squared. This just shows kind of how it works slowly, and so on. This is the lower bound, which is indeed getting better. So I think what we'll do is we'll quit here, and we'll sort of finish this up next time, which will be next Thursday.

[End of Audio]

Duration: 74 minutes

ConvexOptimizationII-Lecture08

Instructor (Stephen Boyd):Okay. Well, well. It's the good weather effect, the post-project proposal submission effect. So this is maybe a very good time for me to remind you that even though the class is televised, attendance is required and we're actually not kidding on that. So it's not cool. We get up early. I guess I can't say "we" since the TAs aren't here. But the last email I got from Jacob was 4:00 a.m. So he was working on lectures for you, so I think we can excuse him. So that's required. Also, I need to tell you guys a very terribly sad story, and the story is this, it transpired, actually, in this very room right here yesterday. So you will recall there's a section in this class, and that the section was devoted to informal working on the projects and things like that. And so in this very room yesterday afternoon, not only the professor, but both TAs showed up. And now I'd like you to guess how many other people showed up. Let's take some guesses.

Student: Zero.

Instructor (Stephen Boyd):That would make a better story, in fact.

Student: Two.

Instructor (Stephen Boyd):No. We can do this bisection now. It's an integer. Well, I guess that's not given, but it's an integer and it's more than zero and less than two. So one person showed up. So by the way, we've already graded their project – actually, we wrote the code for him. We assigned his grade on access. So it was very, very sad, really. So let me just remind everyone that attendance is required, even though we're on video now. The flip side is this, there's a lot of other people watching this other places. So I have to tone down what I say about other things. I just realized that. Now I know there's plenty of other people watching these things. So let me say something about the project proposals. We have looked over about half of them, or something like that, and what we're going to do is if we don't like your project proposal, we're just going to send it back to you and you're going to keep revising it until we're happy. We'll – some of these things are just LaTeX here. That's unacceptable. And by the way, a lot of them are really nice, just what we wanted, clean, LaTeX. We gave you templates and the basic thing is if it looks anything different from what I write, it's not cool. We're not going to accept it. We're just going to flip it. If you've invented some secret new notation or if you look at it – we also saw some total amateur crap in there, like random capitalization of things. This is so uncool it's not even funny. So we're just going to throw these things back at you until you produce a project proposal – if it's more than three pages, we're just going to send it back. I mean, things like that. So actually, now, the good news is we saw a lot that we liked. And so – we may even have those done or at least first responses, because we're not going to spend a lot of time on them if we don't like what we see – we're going to get some first responses back maybe even by tomorrow or something.

And we'll tell you that by email. So let's see, we'll finish up ellipsoid method today. We'll finish up actually the first entire section of the class, which is methods for non-differentiable optimization. And then we're going to move on to the second chunk of the class, which is very, very cool material. It’s going to be distributed and decentralized optimization. So this is really, really – that’s really fun stuff. Okay. So let’s do ellipsoid method. I think ellipsoid method, you remember from last time, is you don’t have to say anything other than this picture is it. You don’t need anything else. There we go. So this picture – why would that be? Okay. This picture says everything. You don’t need to know anything else about the ellipsoid method. Basically says you take an ellipsoid like this, in which you know the solution if it exists lies, you evaluate a subgradient or a cutting plane at the center. You call a cutting-plane oracle. And that eliminates a half space. It says that the solution for sure in not in this half ellipsoid. Now you now it’s in this half ellipsoid. And what you do in order to keep a constant size state or data structure that summarizes your ignorance – if you look at the inside or outside of the ellipsoid. To do that, you put the minimum volume ellipsoid around that half ellipsoid and that’s your new point. So that’s it. And I think we looked at these and we looked at the example in detail. The formula is here. In fact the interesting part is the only thing you do is take a step in a scaled gradient direction, and the scaled gradient direction, you scale by a positive definite matrix that you update. And in fact, what that matrix does at each step, you actually dilate space in the direction of the gradient. So the original length of the gradient ellipsoid method is – well, I don’t know what it was, because it’s in Russia.

But when it’s translated into English, it’s something like subgradient method with spaced dilation in the direction of the gradient. That’s the original full title translated into English. Actually, I’ve seen what the Russian is like, and it looks long, too. That’s what it is. Okay. We talked about the stopping criterion and then this gives you your full ellipsoid method. Compared to a subgradient method, it’s an order n2 algorithm, because if nothing else, you have to write all over p. So it’s an order n2, plus of course, a subgradient evaluation. That might cost more. It’s order n2 to do this. And in fact, let’s see, and of course, the storage is now n2. In the subgradient method, you have to store something like n. So it’s not too relevant and I can tell you why. Because this thing is so slow that running this for more than 100 variables is going to take 50 years or something anyway, so there’s not point really worrying about any storage or anything like that. Okay. We talked about that and we looked at an example. And I want to just mention a couple of improvements. Improvements just means a handful of lines to make this thing work better. One is, of course, it’s not a decent method, so you would keep track of the best point found so far. And of course, at each step you get a lower bound, and that lower bound is the function value minus this. Now remember what this thing is. This is the minimum of the affine lower bound on the function over the current ellipsoid. So this is a lower bound on the optimal value. And this is not increase. Your lower bound does not increase, therefore, you keep track of the best one. That’s lk. And you stop when uk minus lk is less than epsilon. Now, when you stop, this is a completely non-heuristic stopping criteria. It’s completely non-heuristic because u is the objective value obtained by a point you’ve found, and l is a completely valid lower bound. And so this is totally non-heuristic stopping criteria.

Now, I don’t know that this really matters much, but just because the ellipsoid method’s not really used that much, and I’m not even sure this is that much of an issue. But since you’re doing down dates, so a down date is when you subtract – you take a positive definite matrix and you subtract. In fact, this is rank one down date. Now, when you do – and I guess that's supposed to be clever, because if you get a new measurement and something like least squares or something like that, you would do a rank one update. And there would be a plus here. So this is a down date. Down dates, they're fine. In theory, this new matrix will be positive definite. That follows immediately. But because of numerical round-off and things like that, what'll happen is this will become indefinite. Now, there really wouldn't be any problem with that unless your code actually has a square root here. If it has a square root there and you take the square root of a negative number, it depends on what you're using. In a real system, some exception would be thrown. In a not real software, then you'd get some complex numbers and you'll never recover. Of course, that's silly. All you have to do is write is g transpose pg is less than epsilon2. That's all you have to do. And that's actually pretty cool, because it'll also stop when the ellipsoid gets numerically flat in one dimension. So that's what that is. However, there's better ways to if you actually care about this, you can actually propagate a Cholesky factor for this and there are order n2 Cholesky down date algorithms and things like that that work very well. Okay. So here's the same example. And you can see your lower bound and your upper bound slowly converging. One thing you can say here is that you actually converge much faster than you think you have, or know you have, I should say. So the lower bound just take a long time to catch up with the upper bound. Okay. Now we'll get the proof of convergence.

This is going to look completely trivial. Fits on one page, two pages, I guess. But you have to remember that when this was applied to linear programming, in 1979 or something like that, which was the first time anyone proved polynomial time solvability of linear programming, this was a really big deal. And so that's always the fact that things after the fact look simple, especially when they're expressed clearly. It doesn't always. So here the assumptions. We're going to assume f is Lipschitz. So you have a capital G. And G, by the way, is also – an equivalent statement is to say G is the upper bound on the norm of the subgradients. And we'll start with a ball of radius capital R. And now what we're going to do is exactly like subgradient method, the proof is going to go like this: we're going to say that suppose that through the kth step, you have failed to find an epsilon suboptimal point. So epsilon is some positive tolerance, and you have failed to find an epsilon suboptimal point through the kth step. That means the alpha of xi is bigger than f* plus epsilon for all those steps. Now, this is the subtlety. It's actually quite simple and it's this. This says – let's think about what happens when you throw points out. When you throw points out in the subgradient method, it's because an oracle told you – let me draw a picture here. It's because you have a point here, an oracle gave you a subgradient, and what happened was you threw out all these points. Now, there's another step in the ellipsoid algorithms. There's the good step, that's when you throw points out, and then there's the bad step, which is when you actually cover what you know. You actually intentionally increase your ignorance by covering it with ellipsoid. But when you do this, this is the subtlety. Every point you have thrown out here has a function value, which is greater than or equal to the value there.

That's the definition of a subgradient. Well, that's an implication of the subgradient inequality. Now, if this point here is not epsilon suboptimal, so that's not epsilon suboptimal, then every point here is not epsilon suboptimal. Because the function value here is bigger than f* plus epsilon, and therefore, the function value of every point discarded is bigger than f* plus epsilon. That means – here's the epsilon suboptimal set. There it is. That means that in every step of the ellipsoid algorithm up to and including k, you have not thrown out any point in that epsilon suboptimal set. Every point you threw out was worse than one of the function points you evaluated, and all of those were not epsilon suboptimals, so they're over here. So that's the argument here. If you're an epsilon suboptimal point, you are still in the kth ellipsoid. So the kth ellipsoid surrounds this – if that's the epsilon suboptimal set, whatever the kth ellipsoid looks like, it covers this. Okay. So that's that. Now, from the Lipschitz condition, if you are within a distance epsilon over g – let's say an optimal point, actually any optimal point, then you're epsilon suboptimal. So what that says is there's a ball inside this epsilon suboptimal set here. There's a little ball whose radius is epsilon over g. See if I did that right. So there's a ball. So now, that says the following: that says, obviously, if this thing is inside that, its volume is less, so the volume of that little tiny ball is less than or equal to the volume of ek. But these things we know. So the volume of this ball is alphan. This is the volume of a unit ball Rn. It's actually not going to matter because it's going to cancel, but you know it's got gamma function or factorials and stuff in it. Doesn't really matter, anyway. But it scales like epsilon over g to bn. That's less than or equal to this thing. Now, we've already discussed that. The ellipsoid method reduces the volume – at each iteration, the volume of the ellipsoid goes down by the factor at least e to the minus one over 2n. So it goes down by at least one over 2n. So if you do k steps, it's less than this thing. Okay. But this thing is alphan times Capital R to the n. Now you cancel the alphas, take some logs, and you get this. So there it is. One page. That's it. It basically says the following: it says that if you have not produced an epsilon suboptimal point then the maximum number of steps you could possibly have gone is this number.

Another way, you turn it around and you say this: if you go more than this number of steps, for example, this step plus one, then you must have produced an epsilon suboptimal point. Therefore, f best k or whatever it's called, that f best thing that you keep track of the best one, is less than epsilon. That's it. It's simple. It's actually like subgradient – I think it's even easier than subgradient, because there's almost no – there's just nothing there. It's embarrassing. So it's 1975 you would apply this to linear programming and get all the calculations involving the number of bits and all that stuff. You'd be very famous, or something. It's that weird? Okay. So the picture, which I've drawn poorly anyway, is this. You have your suboptimal set. That's an optimal point. You have the little ball here. That's it. All right. We've completed this. Okay. So actually we can interpret that complexity bound. And let's see how that works. So you start this way. You start with e0, that's this initial ball. And of course, you have prior knowledge is the Lipschitz constant, so if someone – before you've done anything, you haven't even called the oracle. You simple have a ball of radius R. And someone says, "Please tell me, what's your best guess of what the point is?" Well, you would just take the center. If someone said, "How far off are you?" The answer is you're off by at most GR. Because the optimal whatever it is, it at most R distance away. And the Lipschitz constant is G, so in the absolute worst case, you can't be more off than GR. So that says that if someone asks you what's f*, you say it's between this, because it's got to be less than that. And that's the smallest the optimal point could be. So essentially, your gap is GR. Not a great gap. Well, it depends on the problem, of course. Now, after you run k iterations, your gap has been reduced to this thing, and we know that that's less than epsilon, where epsilon and k are related by the 2n2 log GR over epsilon.

So what that means is this, it says that the iterations required to take it from your prior uncertainty in f* to your posterior one is exactly this. That's RG, that's your prior uncertainty, and epsilon, that's your posterior uncertainty. And it turns out it means that you need – this is the number of iterations required, 2n2 log GR over epsilon. If you convert this to a log base two here and do the arithmetic correctly, which I may or may not have done – let's suppose I did – then this would tell you that the amount of information you get per oracle per subgradient call is 0.72 over n2 bits per gradient evaluation. In other words – now of course, you'd like the n not to be there, obviously. And in fact, with the CG algorithm, it's not there. It's not. You just get some fixed number of bits and you'd have to work out what it is, like one over log – whatever it is, you'd have to work out what that 0.63 thing is and all that. But the point is, you get a fixed number of bits in the CG method. Ellipsoid method is very simple and implementable. It degrades with n2, with the space dimension. But the wild part is that's a polynomial in all the problem dimensions and stuff like that. So it's really actually – your complexity theory, you'd be jumping for joy at a result like this. And they do. Now unfortunately, the ellipsoid method is, indeed, quite slow. So and that was actually a lot of – obviously a lot of people in the West were also trying to establish the complexity of linear programming, and the Russians. Not only do the Russians do it, but they kinda did it in the '70s and somebody else just noticed – in the early '70s – it had solved some allegedly open problem in the West. You can imagine it wasn't super well received here. And so then people quickly pointed out – they said, well, that's totally useless. Just what you'd expect from the Russians.

They said the simplex method works excellently, but it has exponential worst case complexity, but the ellipsoid method has this nice bound, but works terribly in practice. And they obviously totally missed the whole point of it. Then they embraced it. It was on the front page of the New York Times. Okay. Actually, the very cool thing about it is I really do think you can argue that these methods are actually generalizations of bisection. And that's a pretty weird thing to say, because bisection, when you normally think of it, it's very special. It has to do with R and ordering and stuff like that. If you're in this interval, you check in the middle and at each step your localization region goes down by a factor of two. So it's very difficult to imagine how on earth anything like that could work on multiple dimensions. And you can get very confused thinking about it. The claim is, the ellipsoid method is it. And so the good news is, something like bisection, or bisection in spirit, it is quite like bisection in spirit. You get a constant number of bits on improvement in information per subgradient call, period. It's not one bit. And it degrades with dimension. But it's absolutely constant. And that's kinda cool. So I would claim that that's what this is. And notice that we're off on our bounds here. If you plus in n equals one, this says you get 0.72 bits. We know, in fact, you get one. Now, that's because that e to the minus k over 1 over 2n is actually a bound. The actual number is for n equals one. It's one bit. Now we're going to talk about deep cut ellipsoid method. And deep cut ellipsoid method works like this. Let me see if there's a picture. No. I'll draw one. Here's a picture. Deep cut ellipsoid method works like this, so here is your ellipsoid. Here is your current point. And you evaluate – you call a cutting-plane oracle at that point.

Now, a neutral cut would come back and tell you something like you can forget about everything up here. A deep cut comes back with a bonus of information and basically says something like this, it makes a cut down here and says you can actually forget about everything here. Now by the way, if you're super lucky, and this cut comes outside the ellipsoid, you can say it's not feasible or it's impossible in this case or you go to your reliability system and decrement the reliability on that oracle. So this is a deep cut. And now the idea is this, when you have a deep cut like this, you do the same thing, you put – I don't even dare do it. You put an ellipsoid around, I guess you would call this a less than half an ellipsoid. So that's what it is. It's a less than half an ellipsoid. And sure enough, there are formulas for that. And those formulas work out this way. Actually, they're quite interesting. They're fairly simple. You take alpha is h over this thing. This is the length of h the offset in this new metric. And if that's more than one, the intersection is empty. So that's less than one. And what this says is you actually go in the exact same direction. You step in the exact same direction as the normal ellipsoid, but you take a longer step length, and you actually are more aggressive in your down date. That makes sense. You're getting more bits of information. So this is the deep cut ellipsoid method. By the way, other people have worked out formulas for things like parallel cuts. That's a weird thing where what comes back to you is a slab, a parallel slab and you worked out, actually analytical formulas for the intersection of a slab with an ellipsoid. By the way, these are not attractive formulas, as I'm sure you might imagine. Variations on that. Now the fact is, these are a little bit disappointing because you think that's cool if I'm cutting off more than what I need. And you can always do a deep cut if you have a subgradient evaluation using f(k)best, or whatever.

Because if you already have a value optimal, you get a subgradient if – unless you are right now at the point that was the best one. But you saw, of course, it's not by any means a decent method, so very often, your current point is worse than the best one you found so far. Therefore, an ordinary subgradient call is going to give you a deep cut there, period. You might imagine that by cutting more volume out at each step, you'd do better. And the answer is, well, I guess so. Certainly the volume is lower. But it turns out you don't really do that. It doesn't really enhance the convergence that much. This is not atypical. Quite typical. I mean, you can find examples where it does better, but it often doesn't. Okay. How to do inequality constrained problems. This won't be a surprise to anybody. I think once you get the hang of all these cutting plane methods, it's quite simple, you do this. If x(k) is feasible, then you update the ellipsoid this way, and this is a deep cut here. So that's fine. You do a deep cut thing. If it's infeasible, what you do is you choose any violated constraint here and you do a deep cut here. And a deep cut is with respect to zero. Because basically, what it says when you do this deep cut is any point that satisfies this is guaranteed to have fj bigger than zero, and therefore, to be infeasible. Therefore, anything that satisfied this inequality – sorry, it's the other way around – anything that violates that inequality is absolutely guaranteed to be infeasible. In fact, quite specifically is guaranteed to violate the jth inequality. That's what this means.

So it’s the same story. Once you get used to – it’s kinda boring. Okay. Now is x(k) stays feasible, you have a lower bound. If x(k) is infeasible, then you get this lower bound on f. If this thing here is positive, then that says the whole algorithm can grind to a halt and you actually have a certificate of infeasibility. So it can’t work. You can stop. That’s the stopping criterion.

I also mention this, but I think, again, once you kinda get the idea it’s not a big deal. And it turn out in this case, the epigraph trick doesn’t really help very much. And we will – that finishes up our discussion of the ellipsoid method. And this finishes up, in fact, the first chunk of the course, which is direct methods for handling non-differentiable convex optimization problem directly, using subgradients and things like that. And I guess the summary is you have subgradient methods, which are amazingly stupid one line of algorithm, and a proof that’s about a paragraph. Then you have the ellipsoid method, which is three, four line algorithm. And the proof is two paragraphs, maybe. The ellipsoid method, actually has, depending on exactly how you define what it means to be a polynomial time algorithm, actually has polynomial time complexity. I mention that because when you get into real problems involving real variable, there’s actually different definitions of complexity for the problem. We don’t have to worry about it. The tradition one uses rational inputs and has a number one, which is the total number of bits required to describe the data and all that kind of stuff. People have moved toward a more realistic one, where you it’s a polynomial to get epsilon suboptimality. The number of steps required is – or the number of operations required, whichever you’re counting – is a polynomial of the various dimensions and the log of one over epsilon. Oh, and a magic number that you’re not supposed to ask about. And if you do ask about it, it does depend on the data and it depends on how feasible the problem is. But it’s not considered polite to ask about that. Okay. This finishes up this whole section. I do want to put it all in perspective and remind people that these are extremely bad methods for solving problems. If there’s any way you can pull off an interior point method, then these are a joke in comparison. So these are not – any method where you could even possibly think about using an interior point method, do not use anything like a subgradient method. It makes absolutely no sense. So we will see, actually, the next section of the course is going to be exactly a method where barrier methods cannot be used. And we’ll see why. It really has to do with compartmentalization. And so you should really think of all this as useful when you have literally just a black box or oracle that evaluates subgradients or cutting planes. And for various reasons, legal, confidentiality, whatever, you can’t even look inside. So that’s the best way to think about it.

If you could look inside, you can probably code up everything in terms of a barrier method, and you’ll have something that is so much faster, it’s completely insane. So you have to think about this, there’s some requirement that there’s an interface, which is subgradient oracle, and you’re not allowed to look on the other side. Okay. So now we’re going to start the next topic of the class. It’s an organized – and this is on decomposition methods. And honestly, this is the one that’s really going to, for the first time, show you real examples of problems where you would use a subgradient method. As I said, there have to be compelling reasons to use it. These are going to be very compelling. So let’s just jump right in. I should say these methods are widely used now. They’re – it’s a big deal in networking and in communications. It's a lot of the basis for most of the intelligent thinking about cross-layer optimization in networks. And it has a lot to do with distributed design, distributed methods and things like that. We'll look at these now. I should say there's lots of other places where it's actually used. Okay. We'll start with something really stupid. Here it is. I would like you to minimize f1(x1) plus f2(x2) subject to x1, C1 and x2 and C2. That's the problem. And how do you solve that? Well, the way you solve it, you can solve it this way, because the first group of variables is not in any way connected to the second group, you can choose them independently. And obviously, a sum is minimized by – if they're completely independent – by minimizing each one separately. So here you do this separately. Now, I do want to point something out. By the way, it means you could do it on two processors if you had two processors, for example. That's one possible advantage. And you replaced the sum of the run times by the max. That's one. So you can trivially analyze it.

Now the other thing is, let me ask you this. Let's just make this super-duper practical – let me just ask, let's suppose you don't recognize that your problem is separable. Let's make this super specific. Let's suppose that you take your problem and you type a CDX specification in, which is separable. What will happen? And you can figure out the answer. What should happen is that you get a note saying, "Head's up, your problem was separable." And it could said, semicolon, the next line says, "I'm solving two problems separately." And if you have multi cores, you'd look and some process would be bored and start up on some other process. Maybe something like that should happen. It doesn't. So you tell me, what happened? In fact, let's make it really simple. Let's forget the constraints completely. No constraints. And let's make f1 and f2 smooth. So you're going to use Newton's method. What happens if you accidentally fail to recognize that your function is a sum of two independent functions and you use Newton's method? What happens?

Student:n1 plus n2 [inaudible] that is squared, I think.

Instructor (Stephen Boyd):Squared of what?

Student: Cubed maybe.

Instructor (Stephen Boyd):Cubed. Here's what happens. If you're doing Newton's method, your dominant effort – let's forget actually forming the Hessian and gradient – the dominant effort is going to be something like n1 plus n2 quantity cubed. But if you recognize it as separable, it's n1 cubed plus n2 cubed. Everybody got that? And let me tell you, that second one is a lot better, depending on – well, it depends, actually, on the numbers. If they're very skewed in size, it's not that much better. But if they're equal, it 4x or something. Actually, honestly, 4x is not a number worth going after. But fine. Make it separable. So big payoff in doing Newton's method and you recognize the problem is separable. But let's think very carefully about it. Can you tell me about the Hessian of a function that is a sum?

Student:[Inaudible].

Instructor (Stephen Boyd):That's exactly it. It's block diagonal. Now, that's interesting. So let's say that your code is written well enough that it handles, let's say, sparse matrices. Let's imagine it worked. If it weren't, we already know you're going to pay for it because you're going to get the n1 plus n2 cubed versus n1 cubed plus n2 cubed. We know you're going to pay big time if you don't exploit it. If you're linear algebra is what I call intelligent linear algebra, as opposed to the stupid linear algebra, then that's no reflection on you. So what happens? If you're using intelligent linear algebra and you see something like an h backslash g, let's say, and h is blocked diagonal with an n1 and an n2 block, how long does it take to solve that?

Student:n1 cubed plus [inaudible].

Instructor (Stephen Boyd):You got it. So basically, any sparse solver would recognize that. Any method for sparse matrices will take a matrix which – if it's solving x equals b, will take a matrix a that's blocked diagonal and basically do an ordering that solves one block and then another. Now it's going to do it on one processor, typically, so you don't get the parallelisms speed up, but still. So the conclusion of this story is the following, well, it's a little bit subtle. It basically says that if sparsity is recognized and exploited, the speed-up will come for free. And now we come back to the question that started all this. You have CBX, so you're running CBX, and you throw at it a problem that is separable. Now you tell me what happens.

Student: It's smart enough to know that it's sparse.

Instructor (Stephen Boyd):Absolutely. So it's going to – often. This is subject to the whims of the gods that control heuristic sparse matrix ordering. But unless you wrote some code that made every effort to try to confuse and make a very complicated thing. But if you wrote out a basic one, CBX will very happily compile your problem into a big, let's say second-order program. It will be separable. Of course, you could check that instantly in a pre-solve. But let's forget it. Suppose you don't, it will very happily solve it. And it will say, okay, let's see, what do I have to do at each step? Like, I intend to take 20 or 30 or whatever the number integer point steps is. Say, well, I have to solve this for this primal dual search correction. It sets up a big system of equations and then you make a small offering to the god of sparse matrix ordering, heuristic orderings, and if you did that, and she smiles on you and your problem, then you'll get an ordering that will basically give you the full parallelism speed-up. That's what happens. So the conclusion – that was a very long way to say the following: it probably, except for problems with not recognizing ordering or something like that, it will probably give you the speed up. Okay. This is a long story. So this is – and obviously this is max of f1 and f2, it would be the same thing, that's totally obvious. Okay. So far nothing I've said has been anything totally obvious. It's along those lines where you can't imagine the things we're saying are so trivial that you can't imagine we're headed towards anything of consequence. Actually, most things are like that. They sneak up on you. And now suppose two problems – now we're going to talk about the idea of having a problem that's almost separable. So almost separable would be a problem like this. I want to minimize f1 (x1,y) plus f2 (x2, y). That's my problem.

And here I partition the variables into three groups, the – if you want, you can think of these as the private variables associated with system one, the private variables assisted with system two, and then the joint variables. And by the way, there's no reason not to start talking about it now, because to get the flavor of where all this is going, what the applications are, you might think of this as two portions of a firm, two subunits. That's unit one and unit two. These are choices unit one makes in its operations. And they are private. x2 are the choices that subunit two makes. y are the ones that sort of involve both. So things where they absolutely have to coordinate. Now what you call here is y has lots of names, but it's called the – by the way, a lot of this material goes back to literally 1960. So decomposition is a very old idea. So y is called the complicating variable here because – well, obviously, if it were absent, then this problem would be separable and it would split. So let's see, here you should think of these as the private local variables and you should think of y as a public or an interface or a boundary variable. So these are the names and ideas you should be using to think about how this works.

So here's primal decomposition. I'll give the method. It goes like this: what we're going to do is the following. We're going to have two sub-problems. And what we're going to do is this, you fix – imagine y is fixed. So y is fixed and we're going to minimize over the x1 variable, f1 (x1, y). And we're going to minimize over x2 f2 (x2, y). Okay? The optimize this one we're going to call phi1 and the optimizing of this one we're going to call phi2. These are called – notice one thing. These problems can be solved privately. They do not have to coordinate. They don't even have to tell each other what the optimal value of x was or anything like that. They can use different methods. They can be physically separated. They can run on different processors and all that. They just do this. And we think of this thing as a function of y. Now, that's partial minimization. And you'll remember if you have a convex optimization problem and you minimize over some of the variables and write the optimal value as a function, consider the optimal value of the problem as a function of the remaining variables, that function is convex. So that's the partial minimization rule. So these are convex. And in fact, if you were to minimize phi1 plus phi2 over y, this would solve the original problem. By the way, I hope this isn't confusing. The only way it could possibly confuse you is because a big deal is being made out of something that is utterly trivial. So what is being said here is that if you need to optimize something over three variables, x1, y, and x2, you can do it in any order you want. In other words, you can minimize over x1 and consider the result a function of the remaining two, then optimize over the second, then the third. In this case, we can optimize over x1 and x2 separately and first. And then finally optimize over y. So this is completely trivial. This is called primal decomposition, obviously, because we sort of pulled apart the primal.

And so if you want to get a rough idea of how this might work, it would be something like this. You have two – let's make y just a handful of variables. And by the way, the way you should conceptualize this is this, you should think of x1 as a big variable, x2 as a big variable, and y, hopefully, because these methods will be most effective there, y is small. So for example, you might have, let's say, two people designing a circuit. So that's what it is. And what happens is you want to minimize the power of it. And the way you would do that, each of the two sub-components of the circuit has a power, that's f1 and f2. x1 and x2 are giant vectors that give, for example, the devise lengths and widths of something like that of the – basically, the circuit choices in the different parts. So that's what [inaudible] vectors that tell you how to design that circuit. But they have to coordinate on, for example, a handful of variables. For example, if the whole computation has to be done in 100 microseconds, the first block you might have something like they will agree that the first block will complete its computations in 30 microseconds, and then the second block will then take no more than 70. See what I'm saying? So it's how you divide up the requirements of the timing, for example, would be y. Okay. So now the way this might work is the following: that's some private method, some circuit synthesis method is another one. And what would happen here is, in this case, because it's just one complicating variable, which is y, which is the number of microseconds. And it might go from, let's say 15 to 85 or something, microseconds in which number one has to do it. If you make y too small, this guy is in a super rush, because it's been told that the subcircuit has to be unbelievably fast.

So it redesigns it as best as possible. It consumes a lot of power. This guy, however, is very relaxed, says, oh, 85 microseconds, no problem. All the devises are minimum width and blah, blah. It consumes almost no power, but the sum is bad. So this is sort of the idea. So then you'd solve this problem, which is basically you optimize over that one variable, which is how you divvy up the timing. We'll look at lots of examples, but I just want to give you a rough idea of what this is for. Okay. Now, if the original problem is convex then so is the master problem. And there's lots of ways you can solve the master problem. If there's a single complicating variable, then you can use bisection. So that's one way to solve a convex problem in one variable. You could use a gradient or Newton method if the phi i's are differentiable. By the way, the phi i's typically are not differentiable in many cases. Of course, if epsar, it is. In many interesting cases, they're not. And of courses, you could use subgradient, cutting plane, or ellipsoid method. Now, each iteration of the master problem requires solving the two sub-problems and if you're lucky, this can actually be faster than solving the original problem.

Now, there is a tension here, and I should say a little bit about that. Decomposition, if you look in Google and type, I don't know, "optimization decomposition," you'll get stuff all the way from the '60s. And I should tell you that some things have changed a lot. So in the '60s, remember, this is a time when solving 100 by 100 set of linear equations was a big deal and 1,000 was large scale and all that. It's a joke for us now. This is actually before sparsity methods were well developed and stuff like that. This is like iron core, really visualized the time, to be fair to people then. So decomposition in the '60s was mostly used just to solve problems that for us would be laughingly small. I mean, they would be direct methods kind of things. Then it was used because they just didn't have that much memory, their resources were – the only way to solve a problem, what they would call an enormous problem with up to l,000 variables, which of course, is a joke now – that's what War's law propagated forward 30 or 40 years does. It gets a very, very big number, a big factor. So we can't make fun of them. However, I believe – I should say the following: if you are able to collect f1 and f2 in one place and solve it using an interior point method, you are almost certainly better off doing that.

The sparse matrix methods, we just discussed it. They will actually exploit some of the same structure that you would, and they'll do a way, way better job. So if you have the option of collecting all the problem data into one place and solving it, there is absolutely no doubt that's the right way to do it. Absolutely none. So if you want to visualize an application nowadays, where you might want to do this, it would be basically that these are two companies, or two different design teams, or whatever, and they actually have no intension of sharing the details of their design with the other one. They're two subcontractors. They're making two what they call IP blocks, or whatever, or something like that. And they have absolutely no intension – what they are willing to do is the following: if someone, basically the master says, "Design that block to clock in 15 microseconds," to complete his calculations, then they will come up with a design. They actually may not even reveal x1 until you pay. But you don't need to. All you need to do is have an oracle for this. They're going to have a subgradient and I'll get to that in a minute. So they should think of these methods as one involving encapsulation, layering and communication, boundaries between things, where you have well defined interfaces and the interfaces are there, not because it's 1967 and we're trying to solve what would be called a big problem then. It's not for efficiency. It's for organization reasons. So this seems like layering in communications or something like that. Okay. So we looked at this. And here's primal decomposition algorithm. It's really pretty dumb, but you have to understand this first, and then we'll get there. Here's the way it works. I have two sub-problems and it works like this, the master publishes a variable, y.

And it says, this is y. This is it. I want both of you to deal with it separately. So both of the subunits deal with it. They say, okay, no problem. They take these spec or commands, orders, whatever you want to say. And you each one minimizes their function, and they do it being separate, they can be separate, use totally different methods, different processors, blah, blah. And the only thing they're required to do – so the contract here, or if you want to talk about the layer, the interlayer of communication between the master and the sub-problems is this, the master sends y and the sub-problem does not have to even reveal the optimal x. The sub-problem must send back the following: it says what f1 is and it sends a subgradient. Actually, the truth is, in the simplest algorithms it doesn't even have to say what f1 is. So in the simplest method, it doesn't even have to say what power it achieved. I mean, that's a courtesy. But technically, it doesn't even have to. So you have two units and y is a common variable in a company. Somebody says, "Here's y. It's the vector this, this, these are the entries." And they're not allowed to question it. They optimize the thing. Each generates a minimum cost. And they don't even have to reveal what the minimum cost is.

The protocol requires them to return a subgradient of the optimal cost. So that's required. Okay. Let's look at an example. And let's see how this works. Oh, and this is just a subgradient method. You can use any other method would be fine for non-differentiable optimization. So it's an example. You have two piecewise linear functions in R 20. They're both of size 20. And each is the maximum 100 affine. The actual optimal value of the whole problem is 1.71. Okay. And y is a single variable here. So basically, you have two sets of 20 long vectors. Those are the ones associated with unit one or unit two. And you have a single variable y, a scalar, that they have to cooperate on. So okay. And so here, we vary y, and what you see here is the cost of this is the cost of subunit one. By the way, what kind of function is phi1? It’s the minimum of a piecewise linear function. As a function of a variable appearing affinely in the objective constraints. It’s also piecewise linear. However, it looks curved here. And the reason it looks curved, there’s two possibilities. One is that I’m wrong, which would be unfortunate. But that would be fun, people could go back and look at it. Could go viral among my professor friends. Look. Watch this. See something incredibly stupid. That’s option one. I don’t think that’s true. Option two is the following, is I believe what is the case, when you have a piecewise linear function with maximum 100 affine functions in R 20, it’s piecewise linear function. How many little regions do you think there are? I hear the answer and it was correct. Lots. But what I want to emphasize is it’s not just lots. It’s lots and that’s in a big font. Vast. It’s a huge number. So the point, then, is piecewise linear really doesn’t really help us. So I believe this is piecewise linear.

Student: Total function value goes down at each step?

Instructor (Stephen Boyd):No. No, because it’s subgradient method. By the way, if phi1 and phi2 are differentiable – they’re not here – but if they were differentiable, here’s what you could do. If they’re differentiable, you can use a gradient method, in which case – or a Newton method. The Newton method, you have to renegotiate with your subcontractors here to return a second derivative. That might be tough. But you can certain use a gradient method and for sure it would be guaranteed to go down. But not here. So the sum is this and the optimal value of y is somewhere around here. Something like that. That’s the picture. So this is primal decomposition with bisection on y. And, indeed, here the overall – you actually do worse after one step and so on. And you go down and so on. And it’s a bisection method, so this should be going down like that. That’s primal decomposition using bisection. It’s a stupid example. It’s not the point here. It would be the point if, for example – if the two things here were gigantic units that hated each other, but for some reason had to cooperate on a single variable. And they hate each other so much that they wouldn’t even think of revealing the dimension of x, let alone its value or any of the details. Then this protocol will work. And it’s actually pretty cool, because that’s how you should be conceptualizing all of this is just think about it as this is actually a protocol that allows two almost separate problems to come to mutual global optimality by a very limited number of communication rules. Because that’s really what’s happening here. It’s not much. The interface is very well defined. It is very small. What is revealed is very small. Everybody see what I’m saying. That’s the right way to think of these methods. Sorry, in the modern context. Because currently, these methods are not really used. There’s, I know, some examples like multi-commodity flow.

Generally speaking, these methods are not used the way they were used in the ‘60s, which was just to solve a bigger problem, and they just couldn’t fit everything in the memory, which was pathetically small. And so they would pull one in, kinda solve it, pull the other one in, and then adjust some things and so on. Okay, so at this point I want to ask a question, because I want to go back to Newton’s method. Let’s go back to Newton’s method. So let’s see what happens. Let’s suppose we’re doing – let’s see what happens in Newton’s method when you have a complicating variable. And I’m gonna write the Hessian out. So I want to minimize this. I don't know who can see this. I want to minimize this using Newton's method. Now, we've already discussed what happens when y has dimension zero. When I form the Hessian of this, it's block diagonal and if you're not paying attention, you play n1 plus n2 cubed. If you're paying attention, you play n1 cubed plus n2 cubed. But now, what does the Hessian of that thing look like? That's what I want to know. And I'm going to write it in a very specific way. I want x1, x2 and I want y. And here's my Hessian. This is just a mnemonic. It's way out of whack here. Okay. And try to think of x1 and x2 has huge and y as small. Just do it this way.

Let's try out partial derivatives. So for sure you're gonna get lots of x1 interaction. So let's imagine. And you're going to get x2 interaction. Are you going to get x1 and y cross-terms? Absolutely. And I've drawn this totally wrong and it looks bad. So I'm going to draw it small. Those are the x1, y interactions. How about x2, y? Yep. Okay. How about y, y? Yes. And the, of course, it's symmetric, so you get this and this. And now comes the moment of truth when I get to ask you, does that look like anything you've seen before? Please say yes. Thank you. I was actually getting a little bit depressed. Because we went over 364a so fast. I know I covered this is one lecture. But thank you so much. And I don't want to press my luck here, but may I – so, how would you solve this? Just roughly, no details.

Student:[Inaudible].

Instructor (Stephen Boyd):Okay. And what block would you – just tell me the block that you see that's easy to invert.

Student:[Inaudible].

Instructor (Stephen Boyd):There you go. So the block that you would invert, the block elimination gives you an advantage. When there's a block, you can invert easy. What does invert easy mean? Well, it means invert easier than treating the whole block as a big, dense blob. And in fact, look at that. This is block diagonal. And of course we know if this is like n1, n2, easy is because n1 plus n2 cubed is less than n1 plus n2 quantity cubed. There you go. Funny, because that's exactly the separable issue. So the effort to solve this, you don't have to give me the whole number, but if that was a one there, suppose there was just one complicating variable, what would be the effort to solve that problem? I just want the order in n1 and n2. What would be the order. Block elimination, you would form something like a Shur compliment. This thing times the inverse of that times that. That's a Shur compliment. What size would that be if this was a size one here? One. So the Shur compliment is what we generally call a scalar. Solving scalar equations is pretty easy. That's the div, or whatever it is. So basically that costs nothing, and the only thing you have to do is actually solve this thing times the inverse of that times that. That's a Cholesky on this guy and a back and forward Cholesky on this guy back and forward.

And the total complexity is like n1 cubed plus n2 cubed. That's very cool. Because what it means is, let me just summarize what we've just said. It comes back exactly to our discussion earlier about the f1 and f2. This is very cool. It says if there are complicating variables in a function, like in a function you want to minimize, it's almost separable, except for some irritating little variables or something like that. You're going to find Newton's method. And remember this, because this is sort of the theme of how all this works. That says if you don't know what you're doing, which is another way to say it is if you're using stupid linear algebra or if you don't recognize and exploit structure in your linear algebra, you're gonna pay like crazy. You're gonna say nope, not separable. Suppose all you put in there is a reparability detector. It's going to say, nope, everything's coupled. Sorry. You have to choose everything all at once. Everything's coupled. There's nothing you can do separately. You can't do it on two processes. Oh, by the way, you can do that on two processes. I didn't talk about it at the time, but block elimination will run on multi-processors. And in many cases, that's the whole point. So that's another thing. But I didn't talk about that. So here if you don't exploit linear algebra, you're going to pay for it. You're gong to play n1 plus n2 plus one quantity cubed. If, however, your linear algebra is intelligent, meaning it exploits sparsity, if it does it automatically, you'll just get it immediately. You won't even know why Newton is so fast on a problem like this. Everyone see what I'm saying here? So I promise you, if these are – if you write a mat lab script and it's sparse and all that, and everything, this thing will be way fast, and you'll say, wow, that's fast. You may not know why. Of course, ideally, you should know why.

You definitely should know why because if you ever wanted to do something that was real, which is to say, if you wanted to implement it in a real language, you would need to know why and not just say, because something is happening in CVX or something like that. So these are paralleling things here. Complicating variables in the general non-differentiable case you get an advantage. You can get an advantage. But the same thing comes up in a somewhat different way in Newton's method. That's kind of a parallel theme. They're both ways to exploit structure. One at the linear algebra level; one at the mat lab – sorry – my God, where did that come from? That's the kind of thing I'd like to edit out of lecture. That one I'd like to edit out for sure. Anyway, sorry, you can exploit structure at the linear algebra level or at a much higher level using something like this at a very high level. Let's look at dual decomposition and then this is another – it's very cool. It goes like this, I want to minimize this. The same problem before, f(x1, y) plus f(x2, y). What I do is, like everything in duality, you get these weird things where everyone understands basic duality. You describe the Lagrangian. And then you do incredibly stupid transformations of the problem, like if you have f(ax) plus b, you write it as f(y) and below that, you push on the set of constraints. You append. The constrained y equals a equal b. And you think no good can come of that. So then all of a sudden, you turn on the Lagrangian – all the gears grind forward and something happens. The same thing here. You would hardly – of someone came up to you and said, the first step in this problem is going to be to make two private copies for the different subsystems and add a constraints that they're equal. This doesn't look promising. I mean, I have to just say that right out. Everyone agree? This is not exactly – it's like congratulation, you just added more variables and more constraints and the constraints are not exactly what we'd call complicated. Actually, what's beautiful about this is you think of y1 as the local version of what y is supposed to be. And y2 is the other local version. So of course, we have the constraints that they're equal. I have to say, this doesn't look promising as a beginning.

Now, watch what we're going to do. We're going to form the Legran dual of that transform problem. And when we do that, you're going to get f(x1, y) plus f(x2, y) plus, and of course, just the Lagran multiplier, new.

Multiply that by one minus y2. Again, it doesn't look interesting until you look at this. You stare at this for a while and you realize, my God, this one is separable. You can minimize for x1 and y1. It's a sum. You associate this thing with that and this thing with that and it's completely separable. So for example, if you're asked to find the dual function, it can be done separable. The two sub-systems. So the advantage of having two local copies of what is supposed to be a common variable is this, is they don't have to ask anybody how to minimize this thing. So you just minimize it and you get these things. These are related, of course, to the conjugate functions. But then you end up with two completely independent dual Lagrangian minimization problems, and these give you two components of the overall dual functions, g1 and g2. And by the way, computing these basically minimizing this thing and minimizing that thing, these are called the dual sub-problems. They can be done in parallel and completely separately. And if you want a subgradient of g, you need a subgradient of g1 and a subgradient of g2. These are nothing more than y1 minus y2. Because we did that a little bit earlier. And that's amazing because this has a beautiful interpretation. That's the inconsistency, the inconsistency in your local copies. Or another way to say is discrepancy between your two local copies of it. And that's the subgradient of this thing. And new is going to be a price vector. And so here's dual decomposition. We're going to go over this next time, don't worry. But here's dual decomposition. What you're gonna do is the master will distribute – in primal decomposition, the master produces y. And it's basically fixing variables. It's saying, you get this amount of warehouse space; you get this. You must complete your calculations in 15 microseconds; you get 85. That's primal. Dual is this: you don't mention the numbers.

Instead, you give a price. You say, "The price on warehouse space is $13.00 per square foot. And the price on microseconds of delay is this." And then you send them off. And what they solve, what the sub-problems now solve is they minimize this, that's their cost. But they have to decide y1 here. And the way they do that is through – this could be a fictitious or a real charge, that's what it could be. It could be completely fictitious or it could actually be money transferred. That's the cost. That's when the master – the parent company actually says to them, "You run your business however you want." Optimally, of course. Maximizing profit, whichever you want. However, use warehouse space, then we're going to charge you this. That may not even be on the open market. That might be for internal accounting in the firm. Everyone see what I'm saying here? And then they simply say – then they'll use as much warehouse space as they can until the charge for warehouse space eats into their profit, or whatever. Anyway, you get the idea. Okay. Now, the problem with that is if you set a charge for warehouse space, like for you and for you, and then you use some amount, you use some, there's no reason to believe that the sum is actually the total amount of warehouse space we have.

So we haven't – and basically means that it hasn't. So what I have to do is I have to adjust the price. And in fact, that's what this is. So you simply – so methods like this, by the way, in economics, they're very fundamental and they're called well, this is called a price update method. And in fact, they were talking about this. They would call this an externality or something like that. Basically, it says, oh yeah, you can do whatever you want with y1. But when you use y1, which by the way is supposed to also equal this thing – it has an effect on others. You don't care about the effect on others. It'll only be reflected through this kind of market price here, and then this is a price update method here. Oh, and by the way, the iterates are generally infeasible. And in fact, if the iterates are ever feasible, you terminate instantly. You're done. Absolutely done. So if you solve yours and you solve yours and you achieve consistency, you're absolutely finished. Because then – another way to see that is, of course, this. That's the subgradient of g. A supergradient – however people call it. That's a supergradient of g. If you actually achieve consistency, the supergradient is zero. And if the supergradient is zero for non-differentiable concave function, it's optimal. You're optimal and that's it. Okay. So I think what we'll do is we'll quit here and then continue this next time. And we are working on Homework 4 for you. You have nothing to do right now. We'll fix that. We'll quit here.

[End of Audio]

Duration: 75 minutes

ConvexOptimizationII-Lecture09

Instructor (Stephen Boyd):We should make sure the pile of projects is somewhere. Who's got them?

I guess we'll start. I guess the most obvious thing, I guess most of you have figured out by now is we have looked through these preliminary project proposals, and they're floating around so make sure you get yours. Do not throw those away because we didn't copy them and we wanna make sure there's progress, so when you resubmit them, we want to look at the old one again too; and then judge whether enough progress was made to justify our even looking at it again.

So let me make a couple of – the big announcements, and then I have a list of complaints. I just happened to have read all of them this morning. Again, this morning in order, so; but let me say a couple of things about it. They were okay; we marked up a lot of things. You might not be able to ready what we have so you have to be in touch with us on these, so they'll all be resubmitted.

A few were in really good shape and some of them we just sort of, we didn't even get it, so, or there's bad notations, so we just kind of marked it up and then stopped. So what we'll do is I think it's next week, nominally, is that correct? The end of next week it's the mid – what do we call it? The mid-term something or other, so by the end of next week, you'll resubmit. By the end of next week, it should not just be a rewrite. First of all, a lot of them just need serious editing. Some of those, by the way, you could get to us earlier. That would actually not be a bad thing. But by the end of next week, it should actually start having – I mean, there should be a problem. You should have tried some of these things. I mean, these days with things like CVX, very, very easy to actually do these things. It's faster to actually write code than it is to write a Lay-Tech and English description of it that's coherent. So by next week you should have done that. So the way that's gonna work, and we'll announce this on the website, is that you'll be turning in your – I guess the next round of these things next week. If we didn't like your typesetting or something like that, turn that in immediately to us because there's just no reason for us to read things that are sloppy, so okay.

So what I'm gonna do now, I know this is weird, but I'm just gonna rant for about five minutes, then I'll be fine and then everything will be okay and then we'll move on and stuff like that. So here are things. I'm gonna say them once and then I don't, I turn it back in, I don't want to ever see any of these again and anything that comes in from anybody in this class. It actually drives me nuts if I see it from anybody. All right, so I'm just gonna say it once just so it's said and official and on the record. I have a list. So the first thing is capitalization. We found this completely random capitalization. Sections, half capitalized, half-lower-case, unacceptable. So choose a consistent style and stick with it. We don't care what it is, there are many choices. Titles; title of a paper is generally all caps. By the way, you should use our template. We didn't put it up there to not be used. So I know some of you decided to typeset things your own way, don't. Typeset things our way. Your way, it's okay, there are other styles but in this class, you'll use our style, period. It's just easier for us and for everybody involved. So the capitalization is silly. Look at section titles, I mean, and subsections, and there are; this is the standard, and so if you decide that your capitalization is this for a section, the firs word is capitalized and nothing else, fine. Make it that way for every section, okay. I mean, but it's totally unacceptable to have these, and you'll see us mark these things in these things.

The other thing is problem format, so a lot of people did use our templates. Other people decided to typeset things themselves, don't do it. Use our method, so I don't want to see Macs and then S-key and some weird thing where it was right justified or something. I don't want to see your creative Lay-Teching. It should be our format, period. So basically if your project, if the thing you produced doesn't look like the materials we produce for this class, we're not gonna read it. So just know we've marked these – I don't want to see your Lay-Tech things. For Lay-Tech, there are some really basic things I want to talk about. I mean, these are horrible crimes in Lay-Tech. One is to typeset a mathematical symbol. I mean, to use a mathematical symbol and to put it Roman. Like to say yeah, so the variable x, which is just the letter X. Come on, this is amateurish. Some of you did this, not many. Clean this up instantly. We don't want to see it. I mean, that's like so basic. Now the flipside, I'll explain a little bit. So the flipside is this; when you have a subscript or a superscript or something like that and if it represents or derives from English or any other spoken language, then it is not to be typeset in math, okay. So for example, if you want to have a – for some reason you want to talk about some f as some variable and you want to talk about the sub-optimal point, chosen by some scheme you've come up with, and you want it to look like this, let's say, okay. You want it to look like that. That is math, that is English or derives from English and therefore, the only correct way to do this in math mode is this. That's – and then this is math RN; so that's the only way to do it, okay.

So that's the only way to do it. So, and I'm gonna say these once and I don't ever want to see these again. I want to see these looking right. If you look through everything I write and other people who write well; if you look at that, you'll find these things are followed flawlessly. And by the way, if you want to see an example of a good, perfect style, look at Kenuth here, because that's a perfect, perfect – there are no exceptions of any kind in anything he's ever written. It's not exactly, it's not the format I use, but it is flawless. Just take a look at it. You will find zero inconsistencies. For example, he has a rule which I try to follow, which is this: no sentence starts with a mathematical symbol, for example. Let's see, a couple of other things. Okay, if we go to – for the typesetting just follow our format. It – full page, 12-point article. I don't want to see cube column and your fancy thing. I don't want to see a line and everything. I know how to make an H-rule too. We don't want to see it, so just use ours. It's generic and that's fine; and also we have a rule that when you declare a problem, you must say what the variables are and what the data are.

Also, I guess just some basic things. Oh, let me mention one other thing, sorry. Punctuation in equations, so this is something that's actually not done right in a lot of cases, but there's no reason you shouldn't do it. You cannot just have an equation out in the middle of nowhere. You can't finish a sentence and then all of a sudden write y = ax

+ b. You just can't do it, it makes no sense. It's not English, so the correct structure is something like this. If you say well, we have an output y and we have state x and blah, blah, blah and they are related by, then you can have an equation that says ax = b. If that's the end of the sentence, there's a period after that thing because the type of ax = b, the syntactic format is, I mean, in that case it is itself a little phrase and it has to have period after it. I mean, if you want to say they are related by, then equation ax = b, you have a comma, and you can say where a is the blah, blah, blah, blah, blah, that's fine. There has to be a comma there, so this is something – it's just, I mean, I don't see any reason why you shouldn't just write this way. There's just no reason not to.

Okay, oh, back to the – when you state a problem, No. 1 use our format, that will come by using, just following our template. So, you'll follow our template, but then you must say what are the variables. Oh, by the way, another of the most amateurish possible Lay-Tech error is to put a – I mean, this is just complete amateur stuff – is to put a blank line after an equation, which will, of course, in Lay-Tech, generate a paragraph, okay. So there's no reason for us to see things like that, right. There's just absolutely no reason for us to things like this. To see y = ax + b, something like this, blank line and then this, okay. There's just absolutely no reason. This is just like, this, is – in fact, here, let me make it worse. That's the most – we're not even gonna accept this, right, because this is – I mean, that's wrong, that needs to be like that. This blank line is completely wrong; this blank line has a meaning. It's interpreted by Lay-Tech; it's not what you want and all that kind of stuff. So things like this we don't want to see. Oh, the other thing is just like spelling errors. No reason for us to accept spelling errors, right. So Bib-Tech, use Bib-Tech and use it right. So, not only must your references be in Bib-Tech, but it must be done correctly, and I want to mention a few things here, because these are actually things that are reasonable. Let me say what they are. You must use Bib-Tech, don't trust if you go to Google Scholar or something like that or Site Seer and you import Bib-Tech.

A lot of those are amateurishly done and incorrectly done. Read the manual, I mean, read the first page of the Lay-Tech stuff. Read about Bib-Tech. There are mistakes when you get them off of like Google Scholar, and there are some conventions that depend on different, like the Commonwealth English speaking or whatever. So let me show you what some of those are because they're wrong. This is not right, so this is not right; nor is this with the periods, although in British or U.K. convention that's okay. It's not okay in this class. So that actually should be this, AR. The other thing, of course, but again you have to look at this, use this. The other thing is in Bib-Tech you need to know to escape things; otherwise they'll be made lower case. So Bib-Tech will ignore your – it's gonna ignore your capitalization, it'll enforce its own because it knows it can do it more consistently than you can, but it won't know to capitalize things that you don't tell it. So for example, if there's a colon or something in a title and you want the next thing to be capitalized, then this has to be escape, okay. Like that, if you wanted that to be, you're gonna have to escape it. So, okay; so these are – I'm just going over, for those of you who came in late, I'm just ranting because I read 30 projects in a row this morning to review them all. Okay, now, for some of these we never found out what the problem was. We got a lot of background, we heard about this, that and the other thing; we never found out what the problem was. So what we want is we want to get quickly to something that says, here's the problem, and it should be a mathematical format. Say, you can have some stuff; we're looking for this, that and the other thing. The next – we want to see quickly what is the mathematical problem. The idea is that that problem should be understandable by anybody who is in a mathematically sophisticated field. That includes math, applied math, computational mathematics and engineering, anyone in electrical engineering, computer science, statistics. So you want to write high BBC style math, right, that's understandable. Nice accent, understandable by everyone across the world who does math. So it means you can't have – it's not like you're telling a story. Like you say well, we did this, then we added that and oh, and I forgot about this. Did we tell you about this? And also we want this to be smooth. Oh, and also there's these other constraints.

No, you're gonna get to the problem and say – so you can say you were doing portfolio optimization. You can have a paragraph that's English; I view this, I see it in formal XML. So I see an English description that says we want to do portfolio, you have a number of investments, notice this is English, a vary and mean return and the riskiness of the return, they're correlated and blah, blah, blah, and you say the problem is to allocate, to make an investment that whatever, maximizes return subject to some down [inaudible] risk. That's English, then it starts. Let x denote, that's if you following the Kenuth rule that a symbol does not start a sentence. You say, let xi denote the amount invested in asset i, but there should be a problem there that anyone in math, a random person in the math department you can grab and say, read this and they should look at it, read the first part and say, I don't know anything about investments; but they should read that second part, read your thing and know exactly what the problem is. Everyone see what I'm saying? So that's the high BBC style. It shouldn't have any jargon from some field or something like that. It's just readable by everyone. So okay, in a line we never really – in a bunch of the proposals, we never really got that crisp statement. That crisp statement that I can take to, for example, Brad Osgoode or one of my friends in statistics or anybody I want on campus who, or actually throughout the world, who is sort of, is trained in any mathematical field, and say do you understand it? They'd say, well look, I don't know anything about finance, but I sure understand – that problem I understand perfectly. There should be no undefined symbols and so on. Okay, the other thing is that we want –a lot of times we got the problem, but we didn't really get the approach. That's okay, because that was a preliminary proposal for your project, so but when you resubmit, we'd like to hear a little bit more, we want to hear about the approach because you should not be at this point – you should have thought a little bit about approaches.

Oh and then I'll get back to something. Oh, and the other thing I want to say, please never mix the problem statement method and approach, ever. So these must be separated by paragraphs, or even better like sections. In fact, a very good way to do it is to label each paragraph with, in fact, the paragraph label in Lay-Tech. This is actually a very good way to do it, in developing a document. That's typeset in bold, it's a paragraph title, it's like a sub, sub, sub-section, and you write what the paragraph is about. You can later come back and comment all these out as you generate the final product, but if you can't write something like this for each paragraph, it's not well written; and you could say something like the approach, our problem, background or other methods. So, I often start by writing everything this way and then these get commented out in the end. No one knows they're there, but this is a very good way to develop stuff.

So I think this may finish my – that finishes my ranting and ravings. Actually, we think some of the projects are actually going to be – I mean, a bunch of them are gonna be really good, so it should be fun. Just come and talk to us. These minor things, like I said, like Bib-Tech, all that, putting it in our format, misspellings, you deal with that; and then if you want, bring the clean version back to us and then we'll talk. But do that quickly so we can give you some real content. So if we stopped reading because we didn't like your typesetting for example, which I think is a legitimate reason to stop reading, then come back and – fix it and then come back to us and we'll talk about the content and stuff like that. So okay, so that's it. Okay, so that finishes my rant. Now, actually some serious stuff; we finished last night two lectures that may be useful to some people. They're preliminary, we haven't posted the code yet, but we will. One is on sequential convex optimization, so; and we'll post the code as soon as it's clean enough. So this is for people doing non-convex problems. I know there's a couple, Mike for example, or you guys, so read those, and honestly for you guys it would have been better if we'd gotten that lecture done earlier, but its fine. But go and look at that, it should give you some rough idea about the ideas of sequential convex optimization and we have another one that's on MPC, model predictive control. So, and I don't know, I'm not sure that any projects are directly using that, but you might. You could, if you wanted to go the stochastic case, so, but it's related. So okay, okay, don't mind me. We'll continue.

Today is my big office-hours day, so I'll be around all day and please come by, and if you want to talk first, if you want to get me today for content style feedback, before you go and fix all of the Bib-Tech, Lay-Tech, all that kind of stuff, that's fine. Okay, let's continue with decomposition methods; so, okay. So decomposition methods, we looked before. It's sort of a generalization of this idea that if you had a problem that was completely separable like this, you would solve the few parts separately and the idea in a decomposition method is you should imagine a problem where the problem is almost separable, meaning that you can imagine x1 has dimension of 1,000, xq has a dimension of 1,000 and y has a dimension of 5.

And so the idea is you have a problem with 2,005 variables, but it's got a weird sparsity pattern. The sparsity pattern is that only five variables kind of couple across these, so you should see a graph with like maybe two full graphs and like five little edges or something in the middle or five nodes in the middle that kind of connect to both sides. So, and that's a canonical ones. None of the mathematics requires anything, one thing to be one size or another because it all works no matter what, but that's sort of the picture and the question is how to solve this in that case. And we looked at primal decomposition, primal decomposition is simple. You simply fix this coupling variable and you say, and you minimize the first one and the second one; but now they can be done separately because once you fix the y, like I like to think of it in terms of that graph. When you fix y, you are actually sort of removing those nodes as variables and then the graph disconnects and you've got two, let's say, dense graphs and those can then be solved separately once it disconnects. Now the problem is that if you fix y, and you minimize that, minimize that, there's no reason to believe that's the global optimum, but if you optimize over y, you do get the global optimum.

So primal decomposition works by this method, you simply broadcast what y is as a complicating variable. You have a question?

Student:[Inaudible].

Instructor (Stephen Boyd):Yeah, right.

Student: It can be a very complicated variant, right?

Instructor (Stephen Boyd):Right, so that's a very good question. How do you get expressions for fi1 and fi2? So in some applications you really do get expressions for fi1 and fi2. In the vast majority of applications, there's no expression for fi1 and fi2. They're not – in fact, as we move forward, the interesting thing to find out is, how will be access fi1 and fi2? So we're not generally looking for formulas for them. The access to fi1 and fi2 is going to be the following.

We're gonna evaluate fi at a particular point, so it requires only that we evaluate fi and there's one other thing, there's one other method we need to implement, is we need to implement fi.subgradient, okay; so we have to get a subgradient of fi. So in this case, if – let's suppose that this represents something like a linear program or something like that.

So this is a linear program, then in fact this function is piece y, is like piece y is linear or something like that, but it's fantastically complicated, right. Even a linear program with like ten variables and 50 constraints, the optimal solution is indeed piece y is linear, or a quadratic program even. Quadratic program is piece y is linear, but the number of sort of regions it has, so if you want to write sort of an expression for it would be like 1030 or something, but you're not gonna write an expression for it.

What you need to do is, you need the ability to evaluate fi1 of y and a subgradient; so it's very important to think about. We write the symbol, but the question is what do you actually have to implement to carry out the methods. So, and the answer is you have to evaluate and evaluate a subgradient. Those are the two things we have to do.

Okay, so that's a distance, so you see, here's what actually happens. You actually have to minimize this function in calculating subgradient, so that's what you have to do. In particular, you do not have to form an expression. Now, there are cases where you form an expression. I mean, if for example, this is like quadratic problem – by the way if you have a quadratic, pure quadratic unconstrained problem, convex, and you minimize over one set of variables, what's the result in the remaining variable?

Student:[Inaudible].

Instructor (Stephen Boyd):They have a quadratic function. It's this problem exactly right, so this function is quadratic jointly in x1 and y, and then I'm asked to optimize over x1. You minimize a quadratic, which you can do by linear equation, and what is the remaining function? If you minimize over a quadratic, it's quadratic and what is the quadratic form? Now, for a quadratic you can give an expression, because you can give a matrix. What is the matrix? Sheer complement, exactly.

Okay, so that would be a case where there's an explicit form for how to do this. You minimize this thing and you actually form a sheer complement and stuff like that. That pretty much wraps it up. That's pretty much the only case where there's an explicit form.

Okay, this is final decomposition and we looked at an example and we got the dual decomposition and then this is where we'll start today. So, this is how dual decomposition works. It is very interesting. It turns out it's different, but I'll tell you what it is. Again we want to minimize f1 and x1y plus f2 of x2y and what we do is we now split y into two variables, but we insist that they're equal. Now, this is the kind of thing you do in duality all the time, where you make some incredibly stupid transformation and you think, how can a transformation this stupid actually lead to anything good? And anyway, we can see then it does.

So here's what we do. This is, by the way, if you like to think of it, I like to think of y1 as a local version of the complicating variable, y2 as a local version and then this is a consistency constraint. So, if for example, you are sharing area and power, for example, an area and power budget in designing an integrated circuit and there are two subsystems here, then y1 might be the amount, the number of milliwatts that this guy gets. Let's jus make it power, and then y2 over here, is actually also the number of milliwatts this guy gets. So this guy gets some power budget minus y2 or something like that.

So basically what they do is they would both have a version of what this is, and this consistent, this means consistent. Okay, now we do this, we form a Lagrangean and you get this first function plus that. That's the objective plus new transposed times the quality constraint and now what you realize is that L is separable. It splits into x1,y1 and x2,y2 because these separate precisely because we split y into two versions, here local versions and then this thing is linear and so it's separable.

So it splits, and what that says is that if you are asked to minimize L, you can actually split it and do it independently. At two processors, two locations, however you want to visualize why it's good to do it separate. These separate and so g1 you can calculate separately from g2, like this, and we can actually have all sorts of nice meetings of what new is here. We can think of new as a price, for example, here. New is a – we'll see later if we can think of new as a price.

Okay, now the dual problem then is to maximize this thing, and now I have two functions that can be evaluated separately and so in dual decomposition here's what happens. You distribute not a resource allocation to the two subsystems, but you publish your price; new, and then each one uses the price to calculate an action and then that's what it does.

This is actually an – this says if you like, if that's a cost – the news by the way can be either sign, so they can be prices or subsidies. Whatever you want, but what this basically says is, go ahead and use as much y1 as you like or as little. Please, go ahead, but you have to account for it in your accounting and for example, this new transposed y1 says these are the prices, and this says basically you're paying for it and then it becomes a subsidy for the other one. So that's how that works, and the mater, the dual problem manipulates the prices and you can see clearly what happens. If you crank a component of new up, that's gonna tell this guy, that component of y1 is more expensive and then when it resolves this problem, presumably it'll arrive at a point where there's less y1, less of that component of y1 being used. So that's the idea, okay. Now subgradient of either of these is actually nothing but the residual and that comes from stuff we've done before, but here's dual decomposition, and this is using subgradient method, but you could use ellipsoid method, analytic center cutting-plane, anything you want; any method for non-differential optimization. So it works like this, you solve these dual sub-problems, then you update the dual variables. This is the subgradient update and what happens in this case, is at each step you get a valid lower bound on the optimal price, but you don't have feasibility. You only appreciate feasibility in the limit in dual decomposition; so, which is interesting. Actually, if you're ever feasible here, you stop, because then you're off. So, okay. So let me say a little bit about how you find feasible iterates. When you have –at each iteration you maintain an x1 and a y1 and an x2 and a y2, and it's not feasible in general, meaning that you don't have y1 as y2. So basically your two systems have sort of designed themselves separately. They have a variable that is supposed to be the same, but they're not the same and so the idea is to adjust prices to drive the inconsistency to zero, to drive the two to consistency. However, if they stop for a minute, I mean, if I have one electronic sale driving another, for example, this one might design itself assuming it's driving, for example, a capacitive load of whatever, 10 femtofarads or whatever.

This one might design itself with the requirement that it present a load that is 12 femtofarads, or whatever, so they're off. And if you put them together it won't work, because this guy's driving a higher capacitive load than he was designed for. See what I'm saying? So what you do is you just start, you adjust the prices to adjust these so that these 10 and 12 femtofarads, so the loads actually come into consistency. When they hit consistency, they're done, they're absolutely done. When this guy is driving a 10.3 femtofarad load, was designed for that and the other one has designed that sell to present a 10.3 femtofarad load, you're done. That's the picture, okay. Now, what you can do, is when you've got an iterate here, a very good guess of a feasible point is sort of the average, like this. You take y bar, and this is essentially the projection of this pair of infeasible points onto the feasible set, because the feasible set is just y1 = y2. That gives you an upper bound, but it might be this could be infinity in general cases, in which case it's still an upper bound, it's just not a really informative one. So, okay, now you can also do this. If you have a method that says f1, if you have a primal interface to f1 and f2, you can actually do things – like a prima interface says I'm fixing your interface variables, please optimize your internals to minimize your costs. So, if I make the call to f1's primal interface, if I say, please fix – primal interface means this is exactly the interface you'd use if you were running primal decomposition.

Primal decomposition says I'm assigning you this variable. Now you go and optimize your private variables to deal with it. Dual decomposition says; use whatever you like on these interface or public variables. I'm charging you for it for subsidizing before. That's dual decomposition, so if you are actually able to minimize this over x1 with y bar, that's a primal interface, then you can get a better upper bound here. You have to do better because this one kept x1 and x2 the same. Okay, so this is the same old example we did last time. I mean, it's a trivial example, but here it is. So this is now a function of the – this is the price, or the dual variable, and the first system as a g that looks like that, a dual function and the second system looks like this, or this is the first. It doesn't matter, and you add them up and you maximize and I guess the optimum is somewhere around in here like that. So that's the picture. Okay, and this would be dual decomposition using bisection, and let's look at dual decomposition at every step always gives you a lower bound. Looks like the optimum value is a little bit more than 1.7. This is that crappy lower bound where you simply run dual decomposition, average the two y's and see what the objective is and you can see, by the way, that by here you could terminate with a pretty – with this known duality gap.

If you actually have a primal interface to the subsystems, then you can see that actually you could stop after – I guess that's three steps. Essentially, after three steps, you have done two things. You have actually produced a feasible solution to an objective of whatever that is, 1.71, let's say; and you have produced a lower bound from duality that's like 1.705 or something like that. So for many problems that's called quitting time. So, that's how this works. Now, this one would require both a primal – this would require a primal interface to the two subsystems and this requires a dual interface, so this is how this would work. Okay, so the interpretation I think we've already gone over this, but here's one. You can think of y1 as a vector of resources consumed by the first unit and y2 is the one generated by the second and you have to have something like supply equals demand. Then you set new as a set of prices here and the master algorithm adjusts the prices at each step rather than allocating resources directly. That's what primal decomposition does, and the reason you get the minus and the plus is that one is a supplier and one is a consumer. So, you go back over here to look at what happens. Here you go. If you want to know why the sign difference here, like this, it's because one these is going to be interpreted as a supplier of the goods or the resources and the other as a consumer. So one gets revenue by selling it, and the other gets – pays to use it and that's how that works. So this is sort of the – and as you can imagine, this is all very basic in economics and stuff like that. It's not this clearly stated, by the way, in economics, but oops, I'm not supposed to say that. Sorry, it's just a different language. Okay, so that's that, okay.

Okay, now we're gonna look at what happens when we have constraints. And it's not much different here, so let's see how that works. Let's imagine that we have – now, the constraints we can group into two things. These are to be thought of as private constraints, local to subsystem one, and these are private constraints local to subsystem two. And then we – now these are the constraints that couple. Now by the way, in this problem here, these two problems are actually completely separable. They're only coupled through these constraints, because here, I mean, you could make a more complicated, general form where they're coupled through objective and constraints, but just to make it simple, we're gonna couple only through constraints. So this one is literally the shared resource problem. So for example, that's subsystem one, or one unit of a company or something like that. That's subsystem two, another unit of a company, or that's like cell one in an integrated circuit, cell two or something like that or in some system and these are shared resources. This says, for example, the total power has to be less than 30 milliwatts and the total area has to be less than a millimeter square or your total warehouse space has to be this, or some regulatory thing that covers everything. So that's it, and then the real question then is to figure out how to allocate these shared resources between the two systems, right. So for example, the first component of this says that the sum of two numbers has to be less than zero, and the question is well, how do you do that? Do you make this one -1 and this one +1? Do you make this one +5 and this one -5? That kind of thing, that's the question of the resource allocation. Or do you make them just zero and zero, so that's the question.

Okay, all right, so this is the picture, and primal decomposition again, it's kind of obvious. It basically says the following. You introduce a variable t, which is a fixed resource allocation and you simply say, t is the number of resources you're gonna throw at h1. Sorry, at the first subsystem. So you say, okay, you get this much warehouse space, this much cash, this much this and this much power or whatever you like, it doesn't matter. These are your fixed resources and then this sub-problem one says with these resources fixed by the central office, you go and operate yourself as optimally, given these resource allocations. Well, if you give t to this guy, you have to sort of take it away from over here, because the sum are zero, each component. So here it means you put, you take it away over here. Well, take it away if the entry is positive. If it's negative here, you've taken it from this guy and you've given it to this one. So you say minimize this second system subject to this and the t and –t guarantees that if you add these two together, it's less than zero. That's how you've done; you have an explicit decomposition of the resources. You've said you get 30 milliwatts; you get 70, just period. Now it says deal with it. One possibility, of course, is that these are infeasible, because this thing needs more than 30 milliwatts, let's say. That's no problem, because this will simply return the value of fi1 to infinity. So that's simple enough, and the master problem is to minimize the optimal value of this thing plus the optimal of that one over t, and this is sort of the idea here and the cool part is that once you've fixed t, these can be done separately. Okay, so final decomposition works like this. You have a value of t, you solve the two sub-problems, and you have to find the Lagrange multiplier, Lambda. To find a Lagrange multiplier – sorry, a subgradient. I've already given it away. So here, if you wanted to find out a subgradient for this problem – it's absolutely classic. If you stare at this long enough you realize that t is a vector here.

This is actually something you study in 364a. It's the optimal – it gives you the Trainoff curve of optimal cause as a function of these resources. It's convex in t, that's clear; but the other thing is that if you calculate the Lagrange – if you solve this problem privately and then look at the Lagrange multiplier here, for this thing, if you call that Lambda one, that gives you exactly a subgradient of this function. Here you get a Lagrange multiplier and because that's a minus sign, it switches the sign. It's minus. Okay, so find Lagrange multiplier and Lambda one and Lambda two, and I'll give an interpretation of what these are in a minute, and then it says you update the resource allocation, and so that basically says you update t. This is in pure subgradient, you could use anything else. You could use ellipsoid, any of these other things, smooth subgradient, all these things. This updates that. These are usually the easiest to describe, if not the one that works best, but that's it. So, okay, now this gives you a subgradient of the sum and in this case all the iterates are feasible. I mean, that's assuming that these are finite, so if you have a value of t, that works and this is, these are – yeah, when the sub-problems are feasible, it's finite.

Okay, so here we go. This is two problems. Let's see. There's two quadratic functions, the ci are polyhedra and they're sharing two resources here. There are two resources that they have to divide up and this shows the progress of the cost minus p*. So this is as the iteration goes, you can see that they're already – it's quite close to optimal in terms of sharing, and what's being worked out is how these two resources are split between these two subsystems. And here's the resource allocation. They just start both at zero and then immediately the subsystem one, that's the blue one, starts getting more of the resource. By the way, it's beautiful to ask what – I mean, there's a beautiful interpretation of this. Let's make this scalar, just make it just scalar. So t is a scalar, a single resource is to be divided between two subsystems, then Lambda one has a beautiful interpretation. It is the – if it were differentiable, it's the partial derivative of this optimal cost, with respect to that resource. That's what it would be. So if – let's say you're doing minimization and suppose that the – generally, most minimization problems, the actual cost when you really work it out, is dollars. So let's say you're – the units of fi1 and fi2 is dollars and t1 is power, it doesn't matter. I'm dividing power up between two subunits, right. Then Lambda one has a beautiful interpretation. It's literally – it tells you if it's like five, that tells you – let's see if that's right. Would it be five in that case? It would be – sorry; it would be five, and that means that I can do, for every watt of power I'm given, I'll do better by $5 in cost. It's the partial derivative of the optimal cost with respect to the resources allocated.

It tells you the marginal value of power. This one here calculates the same thing and suppose it's three. So if I have two subsystems, you design something, you design something, I allocate power and then I ask if you have to design yourself – please, I don't even ask you to reveal your design. That's x1 and x2, because your competitors and you hate each other. You just happen to be designing subsystems that are going on at the same system. So all you have to do is report back to me what the cost is in dollars of your design, and your marginal utility for power and if you say it's $5 per watt and you say it's $3, then this tells me how I need to shift resources. So, if you can use it more than you can, that's what the five means, at the next step I'm gonna say, okay was that 30, 70? Because now you're gonna get 32 and you're gonna get 68. I don't know if this makes sense, but this is the – I mean, these things, once you think about them – I mean, you have to understand them on many levels. You have to understand them at the convex analysis, convex optimization level, but it's very important to understand these methods also just make complete sense intuitively. They just make sense. By the way, if I overshoot, what'll happen is these will switch and now you need more and then I'll shove back more and that's actually what – that's what these are. That's me going back and forth, basically shifting microwatts back and forth between you and then at each step, your marginal utility is higher and now, but we're out of the third digit, so actually, in that example this is what we call done and then I'd be shifting power back and fort between you. But at that point, your marginal utilities are equilibrated more or less, and that's how we know we're done. Okay, and this is a resource allocation, so the optimal resource allocation turns out not to be – those add up to zero. Do they or –? Thank you, they're two. Sorry, thank you very much. That's two, thank you. I confused myself. Here there's two resources and then the example I was – the verbal example there was one, so, sorry. These don't have to add up to one. This is, you give .5 to this one and then there's a corresponding down here, the one that looks like it's negative and then this one goes like that. Okay, okay.

So let's do dual decomposition for constraints. That works like this. That's very easy. You simply for a Lagrangean and everything just splits out. It's completely separable. That involves x1, x2 and in this case, these are prices and so again, it works like that. If we go back to our scalar example, you're designing a subsystem, you're designing a subsystem. Instead of allocating you 30 milliwatts and you 70, I now say, make a fictitious internal charge in your design of so many dollars per milliwatt, and so – but use as many as you want. You're gonna use five watts, go ahead, but here's the price, so that's it. Oh, so that's how that works, and then what happens is I publish a price for power, and you guys each design your thing, taking this internal accounting into account, because that's what this is and then, I add up and then report to me how much power you use. If it's more than 100 milliwatts, that's my limit. I crank my power up, because too much power is being used and then presumably I then send back you a message which is a dual update message that says, I've updated my prices, redesign yourself. You do that, and then you come in. I overshot on the price, now together you're using 90 milliwatts and so I now reduce the price a little bit and I keep doing that until between the two of you, you're using 100 milliwatts. That's the price update, that's dual decomposition here. So that's that.

Okay, so here's dual decomposition, and it's quite simple. By the way, this is the vector of resources used, and the + is the violation. So this is a projected subgradient method, because here the prices have to be positive. So this is the projected subgradient method, and it works, and of all the other stuff we've done, it just kind of works, so, and it's actually quite cool. It's a price adjustment method and so on like that. So, and here the coordination is being done, so you can coordinate through prices and you can coordinate through allocation and so some people talk about market economies and controlled economies, and this was very popular in the '60s or '70s or something like that; so, to interpret these things in economics. Okay, so again here's our same example, but in this case, we're adjusting resource prices, and this shows the prices on the two resources and we started them off, I don't know, I guess we started them off up here or whatever, and they shot around. It looks like they're converging to whatever it is, .25 and .35, would've been the right places. So if that were the prices, so then – so the idea is if you get the prices right – I mean, this is sort of the essence of duality. If you get the prices right, then independently, the market will clear, meaning it will use – we will not use more resources than we have, No. 1 and No. 2, you'll actually be optimal. So okay, and this the – you get the same sort of thing that in a handle of steps, in dual decomposition you get a lower bound and then this gives you the projected feasible allocation here, and you can see here that by this step, you're pretty much done, for all practical purposes. Okay, and again, this is the same – I won't go through this. So, okay.

So now what we're gonna do is we've talked about all of this with two subsystems, which is actually a little bit boring. It's the easiest, it's the simplest thing to look at, these two subsystems and you have to understand everything for two subsystems first, but now we're gonna talk about general decomposition structures, where it's much more interesting, and next lecture we're gonna do a lot of applications in network flow and I forget what other areas we're doing. The most interesting ones is when you have much more general decomposition structures. So, here's what we're gonna do. I should add that there's no real reason to force economical form. So the way I view all of this decomposition stuff, is it consists of a set of ideas, so like primal decomposition, dual decomposition, and it's best just worked out on the fly. So when you have a problem, if you're doing networking or wireless or cross-layer optimization or something like that in communications, or whatever application you're doing this in, it's best to sit down and simply use the ideas, not the actual formulas, right. Maybe the formulas will fit your case, but generally speaking it's better to just do it, to actually take the ideas and derive it all again from scratch in whatever your particular problem is. But what we'll do here, is we actually will take economical form, not because you should do everything by putting it in economical form, but because we'll actually look at some of these higher order ideas about – and actually, it's where decomposition gets really interesting.

Okay, so here's what we're gonna do. We're gonna have multiple subsystems, not just two. We're gonna have variable and or constraint coupling between subsets of systems, so it's not gonna be described by a graph. It's gonna be a hypergraph; so for example, I'll draw a hyperedge if there is a constraint that links a subset of the system, right. So we're gonna have a hypergraph here, not a graph. Okay, and a hypergraph by the way, an undirected hypergraph is nothing but – hyperedge is nothing but a subset of the nodes. So it comes up in a bunch of variables, but that's all it is. It's subsets of nodes. Now actually, what we're gonna do is we're gonna assume is that all coupling is via consistency constraints. In other words, if they're resource sharing, if they're – any kind of equality constraints. We're gonna make them all coupling constraints and the way you do that is simple. If you have like two, if you're sharing your resource, you simply take the copy of the resource constraint, build it locally and then you have to work the consistency. So, without loss of generality, all constraints are consistency, right. So, and it would be something like this. If you and I are sharing 100 milliwatts, right; then you would have a variable, which is how much of that you think you're allowed to use. I would also have a separate copy of that, which is how much I think he's allowed to use. I will then use 100 minus that number and you'll use that number. If they're consistent, it means that we're actually correctly divvying up the power. So everything comes down to consistency, so all we have to do is drive towards consistency.

Okay, so let's do – a simple example would be something like this. It's already more complicated than the ones we've seen. So I have three subsystems here and I have private variables like x1, x2, x3 and I have a public variable y1, y2, y3 and y4. Okay, so here the private variables are inside these pins and that's to use the electronics analogy. The pins are associated with public variables and an edge actually is – we'll see later what an edge represents, but an edge is gonna represent a constraint, a consistency constraint. Okay, so in this thing you have two simple edges and this structure represents this problem. You have f of x1, y1, so that's the cost function term here. That's the cost of this one, this cost function depends on both this variable and that variable and then the last one is this and then the constraints are, that's constraints local to subsystem one, constraints local to subsystem two, and constraints local to subsystem three, and then they have – I've introduced private copies of the variables and here we insist that y1 should equal y2, and y3 should equal y4. These are the consistency constraints. So without these consistency constraints, everything separates, okay. By the way, the electrical analogy is very interesting and for those who know about electrical engineering and circuit analysis and stuff like that, it's actually quite useful because you could think, and this is again if you don't know about this it's fine, but if you do, you can think of like y1 and y2, if you thought of that as a wire, if you hook a wire between two things, what's equal on the two pins? What potential or the voltage? So you should think of y1 as like a voltage.

Oh, and what can you say about the currents flowing in here and the currents flowing in here? They sum to zero, exactly; so the first one is like ADL and the second one is KCL. Okay, so the currents, in this case you're gonna have to do with the Lagrange multipliers, right, because we're going to expect that. So again, for those of you who don't know about circuits, no problem. Okay, so here's a more complex example, it's quite interesting. You've got like five subsystems here and what happens here is one constraint says that these three variables have to be the same. This one, this one and this one – oh by the way, what can you say about the voltages on these pins in this case? They're equal. And what can you say about the currents entering subsystem one, two and four? They sum to zero, that's [inaudible]. Okay, so we'll get to that. So this is the idea. By the way, this now a quite complicated system, right. It's not a simple thing, it's not given by a block diagonal with little arrow things like that. It's a more complex structure here; interaction structure, that's it. Okay, so the general form is gonna look like this. We're gonna minimize a sum of local objectives, subject to some local constraints, but a local constraint and a local objective can mention two things, it is allowed to mention private variables and it is allowed to mention public interface variables. In fact, the excise don't make any difference at all. They actually just go away because you minimize over them. I just draw them here to remind you that there are private variables, but in fact, you don't have to, you just get rid of the x's. You don't even need them. In fact, at some level, they're just confusing things.

Then you have this, the only thing that couples. This is utterly separable, except for this, consistency constraints, and consistency constraints say this. For every hyperedge I introduce, I'm gonna have a vector of variables, which is in size, equal to the number of hyperedges here and then ei is gonna be a matrix, it's gonna give the net list or hypergraph and it's gonnna tell you each entry, each pin, which hyperedge it's in. So these ei's are matrices whose rows are unit vectors. I mean, this is kind of a silly data structure for it, but still, it's good enough. So, I mean, it's fine, actually. It' just tells you, it's a note that tells you which – on the y5, the fourth component, which mesh is it in, or which hyperedge it's in. So primal decomposition is gonna look like this. I'm gonna minimize this, I'm gonna fix these things. If you like, I'm gonna fix the voltages, actually, here. I'm gonna fix the voltages and what I'll do is you distribute the net variables to the subsystems, that's what this is. This looks like a matrix multiplier, but it's not, it just distributes the given levels, then you optimize the subsystems separately and you get the – each of these is gonna give you a subgradient here. Then what you do is you collect – if you're working at what the – if you want to get a subgradient of this sum here, you simply do this. You sum – yeah, I transposed gi, it's silly. That's actually – it looks complicated, but in fact it's nothing but summing over the edges. And by the way, this has a – these are the currents flowing into these pins, if you want a circuit analogy. This is the sum of the currents on that subnet. That should be zero, that's KCL. So g is in fact the current, the KCL residual. It's the amount by which KCL doesn't hold. That's what g is, and then it says you update the net variables. So you will update the voltages, like depending on this current. This is the current residual, like that. Again, if you know electrical engineering, that's how that works.

Dual decomposition – and I'll draw actually a picture in a minute that makes this look like a circuit. So dual decomposition looks like this. Instead of fixing the voltages at the pins, I'm gonna actually fix the currents flowing into each pin like this, but the currents are gonna be consistent, so they're gonna be – I'm gonna allocate currents, so when I have a network of three things coming together, I'll say that's +1 amp, -2 and then whatever, let's do +1, here, and they add at zero. So that's the idea, I'm gonna just – I'll make something that's KCL compliant, but now what I need is something like a KVL residual. The KVL residual is – you'll set the currents at the pins, and you'll collect the voltages and if they're all equal, you're in KVL compliance. Actually, in that case, your primal feasible and you're done; but instead you'll actually calculate the average value of the public variable. This looks complicated, it looks like a pseudo inverse, but e is this matrix, it's just got ones on each row. It's a unit vector, so this is just the matrix formula. It is projection; this is a projected subgradient method. Actually, all it really is, is averaging the public variables over each net and then you update the currents that way and I'm gonna draw a picture and we'll see how this works. So this is this bigger, more complicated example and just showing how these things work and I think I won't go into all of this because I guess it's not – I mean, obviously it works, right.

That's kind of – that's obvious, but is kind of cool to actually put these things together and see that they work. I should mention all the source code for this is online. This is actually running, this is a dual decomposition method, and here, this is the residual, this is the total residual, the error in your KVL or whatever, your voltage consistency, or it's just the error in your voltages and that's it. Okay, so what I want I'm gonna do now, is actually, I want to draw the pictures for what these updates are, and then I think that will make it clear. Four people in electrical engineering, it will make very clear. So let me draw that picture. So let's do this, let's have three pins like that and let's say that – let's stick them together like that, okay. So primal decomposition works like this. I fix a voltage on this, I fix a voltage, I enforce a voltage on this net, okay. Now, a voltage says that's basically fixing the value of like y1, y2 and y3. I just fix a common voltage, I say it’s 1.1 volts here and tell each of these to deal with it. So they deal with it and then they will each report a current, let’s say new1, new2 and new3, okay. Now, if the sum of these add up to zero, you’re actually optimal, the whole thing is optimal.

And by the way, in the circuit case it means you’re in equilibrium, okay, so that’s what that means. Okay, so instead what happens is I’ll add up the currents here and, for example, if they’re positive, I will do the following. If these are positive, I will decrease the common voltage, okay, because then less current will flow. I mean, by the way, if you want we can do it both circuits and we can do it in economics. We can do either one; let’s do circuits for a while. The circuit says, and in fact, if you want to know what this is, I’m gonna show you what it is. It’s that, okay. So, let’s work that out of why it’s this. Let’s take a voltage on – well, that’s a wire, the capacitor; there’s a common voltage y on all of these things, okay. As a result, a current flows here, here and here and the total current that flows out of the capacitor is minus the sum of these currents. If that’s zero, done, everybody’s in equilibrium, okay. However, this current – actually, let’s propagate forward in time some appropriate small step. If you propagate forward in time, it simply detriments the voltage off of here.

So primal decomposition is basically running this electrical circuit until it comes to equilibrium. When it comes to equilibrium, you will have the following. The voltage here, here and here will be the same. Well sure, because there’s a wire connecting it, but more importantly, the sum of these three Lagrange multipliers will be zero, and that’s KCL, so for those of you who know electrical engineering, this is kind of the picture. Makes sense? So that’ the picture, okay. You can also do it by economics too. Economics basically would go something like this. I’ll simply from the headquarters of my firm announce what y is, just say y is this. Each subsystem, deal with it. They each deal with it, but they come back with a number that says how much better they could do if they were given a little bit more of that resource, okay. Now, if the sum of those numbers doesn’t add up, it means I could change the amount of resource, I could change the resources and actually do a little bit better, so that’s sort of the idea there. Anyway, so this is the primal decomposition. Dual decomposition I’m gonna do in a different way. I’m gonna do it for the tube case, because most people don’t know about multiple terminal inductors, so if you don’t mind. Now we’re gonna do dual decomposition. Dual decomposition is this picture. In dual decomposition, remember, y1 and y2 are not the same. So here’s dual decomposition, that’s dual decomposition. I claim, and let’s see how this works.

Well, what is fixed in dual decomposition is a current like that and it’s a positive current for this guy and a negative current for that one. So I distribute a current new and one of these subsystems has got a minus sign and the other one it’s a plus sign, okay. Now what happens is they get a current and then they develop a voltage here y1 and y2. If the voltages are equal, we’re done, everything is in equilibrium. Not only that, if the voltage across an inductor is zero, then the derivative of the current is zero; so the current is stationary. Otherwise, what will happen is this; y2 might be bigger than y1. In which case, in a small time later, this one will actually decrease. It will actually decrease. It will have new dots; l new dot is y2 – y1. That’s the current equation. So in dual decomposition you're actually integrating forward this system and you're waiting for equilibrium. At equilibrium, you will get a constant current, which is the optimal Lagrange multiplier and the voltage across here and here will be what at equilibrium? Zero, yeah. So that's again a way to do it. So in this case, you're forcing a current and you're waiting for the voltages to equilibrate, and in the other case, which would be in the two terminal thing, this – the voltage is fixed across them, you have primal consistency and you're waiting for the currents to equilibrate, which is your waiting for dual feasibility or something like that. So I don't know if this makes sense, but this actually a very, very good way to think of these things, okay. Okay, so I think this finishes what we –next time we actually are going to do instead of abstract decomposition, we're gonna do specific applications of decomposition methods and things like that; and as I say for those of you who came in after or during my rant or something like that, I should say that we'll be around.

Please get in touch with us about your projects. Some of them were very, very clean, but think about the things I said, or go back and watch my rant and make sure that none of it applies to your project, but come and talk to us because we now know what they're all about and we'll have constructive comments for all of them. So we'll quit here.

[End of Audio]

Duration: 70 minutes

ConvexOptimizationII-Lecture10

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

Student:[Inaudible]

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

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

Student:[Inaudible]

Instructor (Stephen Boyd):Why block?

Student:[Inaudible]

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

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

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

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

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

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

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

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

Student: Zero.

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

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

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

Student:[Inaudible]

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

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

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

Student: Lower prices.

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

Student:[Inaudible]

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

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

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

Student:[Inaudible]

Instructor (Stephen Boyd):When would that happen?

Student:[Inaudible]

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

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

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

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

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

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

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

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

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

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

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

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

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

Student:[Inaudible]

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

[Inaudible]

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

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

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

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

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

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

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

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

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

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

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

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

Student: Are there other options?

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

[End of Audio]

Duration: 77 minutes

ConvexOptimizationII-Lecture11

Instructor (Stephen Boyd):Hey, I guess we'll start. Let me just make a couple of announcements. I guess we've combined the rewrite of the preliminary project proposal with the mid-term project review, and I think that's due this Friday. Good, okay, it's due Friday. So that's going to be that. Please you're welcome to show me or the TAs if you want a format scan. the TAs are now as nasty as I am, and they can scan something, just walk down it and say, 'That's typesetting error. I'm not going to read any farther.' Or they'll read down a half-page and say, 'What are the variables?' and then throw it at you with a sneer.

So I've trained them pretty well. The danger, of course, is then they start applying that to the stuff I write, which has happened in the past. They say things like, 'This isn't consistent. Use this phrase on this page and this one on the other one.' And you look at the two, and you say, 'Yeah, that's true, all right.' The executive decision was made. We're going to switch around.

It's not the natural order of the class, in fact, but it fits better with people who are doing projects. So a bunch of people are doing projects that involve non-convex problems, and so today we switched around, and we're going to do sequential convex programming first, and then we'll switch back to problems that are convex problems and things like that. We'll do large-scale stuff next.

The good news is that we can actually pull this off, I think, in one day. So we're going to come back later. There'll be several other lectures on problems that are not convex and various methods. There's going to be a problem on reluxations. We're going to have a whole study of L1 type methods for sparse solutions. Those will come later. But this is really our first foray outside of convex optimization.

So it's a very simple, basic method. There's very, very little you can say about it theoretically, but that's fine. It's something that works quite well. Don't confuse it – although it's related to something called sequential quadratic programming. That's something you'll hear about a lot if you go to Google and things like that. Those are things that would come up. But I just wanted to collect together some of the topics that come up.

Okay, so let's do sequential convex programming. Let's see here. There we go. Okay, so I guess it's sort of implicit for the entire last quarter and this one. I mean the whole point of using convex optimization methods on convex problems is you always get the global solution. I mean up to numerical accuracy. So marginal and numerical accuracy, you always get the global solution, and it's always fast. And that's fast according to theorist. If you want a complexity theory, there are bounds that grow like polynomials and various things, problem size, log one over epsilon, which is the accuracy. And actually, in practice as well, these methods are very fast and extremely reliable.

Okay, now for general non-convex problems, you really have to give up one of these. Either you give up always global or you give up always fast. And these are basically the big bifurcation. I mean there are things in between, but roughly, this is the idea. As long as you have local optimization methods, these are methods that are fast. They're plenty fast. They're just as fast as convex optimization methods, but they need not find – they relax what they mean by a solution. So they find some kind of a local solution to a problem. It might not be the global one – can well be the global one, but you won't know it. That's the local optimization.

At the other extreme, you have global optimization methods, and we'll talk about these later in the class. These actually find the global solution, and they certify it. So when you solve the problem, when it stops, it basically says, 'I have a point with this optimal value, and I can prove that I'm not more than 1e – 3 away from the solution period. Right? So now, in convex optimization, we do that very easily with the duality certificate.

So you take a dual feasible point. So when you finish your problem, you say, 'Here's this point. It's feasible, and it achieves its objective value. Here's a dual feasible point, which proves a lower bound, which is epsilon away from the objective of your feasible point, and that's how you prove it. For non-convex problems, the proofs, the certificates as we'll see, are bigger and longer. We'll see what they are.

Okay, now what we'll talk about here is a specific class of local optimization methods, and they're based on solving a sequence of convex programs. It should be understood, but the semantics of everything we're going to do today is. We're not solving any problems at all. When we 'solve' a problem. I'll stop saying quote, unquote, but you'll hear it. I'll aspirate the q and they'll be an air puff after the t, and that's the quotes.

So it has to be understood; when we say we 'solve' this problem or 'here's the result we get,' these are not the solutions of the problems, as far as we know. They might be, and they are pretty good. But it's to be understood that we are now in the world of non-convex optimization, and there may be bounds you get, but that's a separate story.

Sequential convex programming goes like this. It's going to solve a sequence of convex problems, right. And in fact it fits in this big hierarchy, right, where if someone says to you, 'How do you solve this convex problem?' It's actually a sequence of reductions, and the reductions go like this. I just want to put this whole thing in this main hierarchy.

Reduction goes like this. Someone says, 'Well how do you solve this problem with inequality constraints and all that kind of stuff?' And you say, 'No problem. I use a barrier method, which basically reduces the solution of that problem with constraints to a smooth convex minimization problem.' And some one says, 'Well, yeah, how many of those do you have to solve?' And you say, '20.' And they say, 'Really 20?' And you go, 'Okay, sometimes 40.' That's the right way to say it.

Then it's, 'How do you solve a smooth minimization problem?' And you say, 'Well, I solve each of those by solving a sequence of quadratic convex problems.' So that's how that works. That’s exactly what Newton’s Method is. And someone says, ‘Yeah, how many of those do you have to solve?’ And you go, ‘I don’t know, 5 each time,’ or something roughly. And then you say, ‘How do you solve a quadratic minimization problem with equality constraints?’ The answer is, ‘I solve linear equations.’ If they keep going, then you can explain all the methods about exploiting structure in linear equations.

So anyway, you can see that each level is based on reducing the solution of that problem to solving a sequence of the ones below it. And ultimately, it all comes down to linear equations, and in fact, if you profile any code – almost any code – all it’s doing, solving linear equations, nothing else. Okay? So people who work on numerical linear algebra are always very happy to point that out.

Okay, so this fits on top of that hierarchy. So you have a problem that’s not convex and you want to ‘solve’ it. And you do that by reducing it to a sequence of convex programs. Okay, all right. Now the advantage of this is that the convex portions of the problem are handled exactly and efficiently because in a lot of problems – and we’ll see just from our baby examples – a lot of the problem is convex. Huge parts of the objective are convex, lots of constraints; I mean just very typical constraints, just upper and lower bounds on variables. So a lot of these are going to be convex.

And those are going to be handled exactly. That’s the good news. On the other hand, I’ve already said this a couple of times. We have to understand. This is a euristic. It can fail to find even a feasible point, even if one exists. Even if it finds a point and does something, there’s absolutely no reason to believe it’s the global optimum. So I think I’ve said that enough times, but in the context of this course, it’s worth repeating it a bunch of times.

Now, in fact the results also can and often do depend on the starting point. You can even look at that from the half-empty or half-full glass approach. The half-empty glass approach says, ‘Well it shows you this is all stupid and euristics. If it depends on where you started from, then why should I trust you?’ I sort of agree with that. Here’s the half full. The half-full approach says, ‘Oh, no problem. That’s fantastic. We’ll run the algorithm many times from different starting points, and we’ll see where it converges to.’

If it always converges to the same point and the person we’re defending this to isn’t too snappy, we can say, ‘Oh, we think this is the global solution because I started my method from many different points, and it always came to the same thing.’ If you’re an optimist again, if it converges to many different points, you say, ‘No problem. I’ll run it 10 times. I’ll take the best solution found in my 10 runs, and therefore, you see it’s an advantage. It’s a feature that you do this.’

Now actually, these methods often work really well. And that means it finds a feasible point with good – if not optimal, actually often it is optimal, you don’t know it – point. That’s the background. So we’re going to consider a non-convex problem, standard optimization problem, and you can have equality constraints that are not affine, and you can have inequality constraints and so on. And the basic idea of sequential convex program is extremely simple. I mean it’s very simple.

It goes like this. At each point, you're going to maintain an estimate of the solution. So we're going to call that Xk. The superscript K is going to denote the iteration counter. So you can have a counter, which is the iteration. And what's going to happen is you're going to maintain something called a 'trust region.' And a trust region is – I'll get to what it is in a minute, or we'll see how it acts. It's basically a little region, which includes Xk. So it surrounds Xk, and it's the region over which you propose to find a slightly better solution. That's the idea.

So if the trust region is small, it says you're only looking very locally, and Xk + 1 is going to be local. If it's bigger, then it means you're willing to take bigger steps. So that's the idea. So here's what you do. You're given a trust region, and each inequality function and your objective; you ask to form a convex approximation of FI over the trust region. And we'll see. There's going to be lots of ways to do that, some simple, some complex, and so on. And then for the equality constraints, you're going to ask each possibly non-affine constraint here to generate an affine approximation of itself.

Now obviously, if the objective or a bunch of the inequalities are convex, a perfectly good approximation of itself is itself. Right? So in that case, that's very – so forming a convex approximation of a convex function is very easy, same for affine. So I simply replace all of the objective, inequality constraint functions, and the equality constraint functions what their appropriate approximations. And I impose this last condition, which is the trust region constraint. And now I solve this problem. That's a convex problem here. So that's the idea.

And this is not a real constraint in the problem, obviously, because TK is made up by us. So it's not a real constraint. This is here to make sure that these approximations are still roughly speaking valid. That's the idea. So the trust region, a typical thing, although this is by no means – by the way, this is really a set of ideas. So don't – the algorithm we show here is going to be simplified. When you actually do these things, depending on what your particular problem is, you'll switch all sorts of things around. You'll use different approximations, different trust regions. It all depends on the problem. So it's really more of an idea.

A typical one would be a box. Now, if a variable appears only in convex inequalities and affine equalities, then you can take that row phi to the infinity because you don't need to limit a variable that appears only in convex equalities and inequalities. Convex equality, by that of course I mean an affine equality.

How do you get these approximations? Well, let's see. The most obvious is to go back to the 18th century and look at things like Taylor expansions. It's the most obvious thing. In fact, that's all of calculus. That's what calculus is. It's just nothing but an answer to the question, 'How do you get an affine approximation?' Roughly, right? 'How do you get an affine approximation of a function in an organized way?' And the answer is you torture children for 14 years memorizing stupid formulas long after – centuries after anyone has remembered what any of it is for. So that's the answer to that question, how do you get that approximation.

So here's your basic 18th century approximation. If you want a convex approximation, then there's something interesting you can do here. Of course, the second order Taylor expansion would have the Heschen right here. That's this. But if the Heschen is indefinite and you want a convex approximation, you can simply drop the negative part of a matrix. By the way, how do you get the positive and negative parts of a matrix? In this case, it's just talking about expansion. It's the SV, but not quite. It's the ERC iGAn expansion. So you take the IGAn value expansion. You set the negative IGAn values to 0, and that's the projection onto – that's what P is, this PSD part.

Now, these are very interesting. These are local approximations right. I mean that's the whole point of calculus right. That's this thing. And they don't depend on the trust region radii. But that's kind of the idea. So calculus is vague, and it works like this. If you say, 'Yeah, how good an approximation is this?' You say, 'Oh, yeah. It's very good.' And you say, 'Well, what does that mean?' Well, very good means if this is small, then the difference between this and this is small squared. You say, 'Yeah, but which constant in front of the small?' You say, 'I don't know. It depends on the problem and all.' It's vague.

It basically has to do with limits and all that kind of stuff. Actually, I'll show you something more useful, and this is more modern. And in my opinion, this is the correct way. So not to knock calculus or anything, but this is. Actually, the reason I like to emphasize all that is because all of you have been brainwashed for so long, and especially when you were very young calculating derivatives. So that if someone says affine approximation, what's the little part of your brain right in the center that controls breathing and stuff like that?

This comes out just out of that part, depending on when and how long you were tortured learning how to differentiate T2 sin T. Okay, so that's why I like to emphasize that there's new methods. They're relatively modern. They came up in estimation first and other areas. They're called particle methods for a bunch of reasons because they came up first in estimation. I'll show you what they are.

Here's a particle method. This is the modern way to form and affine or a convex approximation, and it goes like this. What you do is you choose some points. There are problems with this, and I'll talk about that in a minute. I'm not saying throw the calculus out. But what you do is you choose a bunch of points in the trust region, and there's a big literature on this. It's not even a literature because they don't really say anything. It's like a lore, and it's like a craft actually. People even talk. I've heard people say things like, 'What'd you use?' 'Oh, I used the sigma points or something.' What's a sigma point or something? And it's some euristic for determining some points or something.

Anyway, so here are the types of things you could use. You could use all vertices, depends on the dimension of the problem. If the dimension of the problem is like 5, all vertices are 32 points. You might do 3. You could do the center and all vertices and stuff because you want a sample of function in a little box. This is actually much better done when each function, even if the terminal variables are large, each function only involves a couple of variables. That's what you're hoping for.

Other ones, you can do some vertices. You can use a grid. You can generate random points. There's a whole lore of this. And you simply evaluate your function, and now you have a pile of data that looks like this. It's the argument and the value of the function. And what you do is, you have a pile of these and you fit these whether convex or affine function depending on what was asked. So that's the picture.

Of course, it's going to come up. That's a convex optimization problem as well, not surprisingly. There are a lot of advantages of this. One is that it actually works pretty well with non-differentiable functions or functions for which evaluating the derivatives are difficult, right.

So for example, if you want to do an optimal control problem with some vehicle or something like that, some air vehicle or whatever, and someone says 'here's my simulator,' and it's some giant pile of code. I guarantee you it will have all sorts of look-up tables, horrible polynomial approximations and things like that. You'll look deep, deep, deep into this differential equation or something, and no one will even know what it is, some function that calculates the lift or the drag as a function of angle of attack and dynamic pressure or something like that, and it's going to be a look-up table obtained from wind tunnel tests. That's what it's going to be.

If it's anything else, it's because someone else fit it for you, but you shouldn't let someone else fit something for you because they didn't care about convexity, and you do. So that's why you shouldn't just let them fit things. The joke is actually a lot of times people fit things, and by accident, the fits are convex. Often that happens.

So this works really well especially for functions for which evaluating derivatives is difficult or given by look-up tables and all that kind of stuff. Now, when you get a model this way, it's actually very interesting. You shouldn't call it a local model. A local model basically refers to calculus. If someone says, 'Here's my local model.' And you say, 'Well how accurate is it?' You can say, 'Extremely accurate provided you're near the point around which it's developed.' So it's kind of this vague answer.

This one is not a global model. A global model says, 'No, that's the power.' That's an accurate expression for the power over three orders of magnitude of these things. That's the power. I'm telling you. That's a global model. I call these regional models because it depends on the region. And so your model actually – let me just draw some pictures here. We should have done this for the lecture, but oh well. We just wrote these.

Let's draw a picture. Could you raise this just a few inches here? I think we'd be better off. Cool, that's good. Even better, that's perfect.

Here's some function that we want to fit. It doesn't even really matter. So if you take this point and someone says, 'Make me an affine model.' Then of course, the calculus returns that approximation, right. So a regional model would do something like this. In a regional model, you have to say what the region is, and the model is going to depend on the region asked for. So if you say, 'Would you mind making me a regional model over – that's too small, sorry. Let's make it here to here. So we want to make a regional model over that range.

I don't know what it's going to be, but it's going to look something like that, okay. Now, here's the cool part. It doesn't even have to go through that point, okay. And you can see now that you're going to get much better results with this, in terms of getting a better point. Now, it also means that those trust regions need to tighten before you can make some claim about high accuracy or whatever. But that's the idea. Is this clear? I think these are totally obvious but really important ideas.

So how do you fit an affine or quadratic function data? Well, actually affine is 263. So it's least squares. I mean in the simplest case it's 263. So affine model is just least squares, and I'm not even going to discuss it. By the way, you don't even have to do least squares. Once you know about convex optimization, you can make it anything you like. So if you want to do mini-max fit, if you want to allow a few weird outliers, throw in an L1 Huber fit or something. So you know what to do. If you don't care about accuracy, if an error of +/- 0.1 doesn't bother you and then you start getting irritated, put in a dead zone.

All this goes without saying, so use everything you want to use. I should add that every iteration in every function appearing in your problem, you're going to be doing this. So is that going to be slow? Yes, if it's written in mad lab, right. But if it's done properly, you're solving extremely small convex problems. If this is done correctly, if it's written in C, these things will just fly. These are sub-millisecond problems here. That's also not a problem unless you do it in some very high level interpretive thing.

This is an example of showing how you fit a convex quadratic. And that you would do this way. It's an SCP because you have a positive semi-definite constraint here, and then this here is, of course, the convex that you're fitting. Your variables are P, Q, and R. These are data, and this objective here is, of course, convex quadratic in the data. So that's a least squares problem with a semi-definite restraint.

Now, another method, which we will see shortly, is quasi-linearization. You can certainly say it's a cheap and simple method for affine approximation. To be honest with you, I don't know why – I can't think of anything good about it except it appeals to one, the people's laziness. I can't think of any other good reason to do this except laziness. We did it later in an example, but here it is. It's kind of dumb. It basically says this. You write your non-affine function as ax + b, but you allow a and b to vary with x. Now that's kind of stupid because one example is just to take this 0 and say that b is h.

As you can see right out of the gate here, this method is not looking too sophisticated. It's not. It's not a sophisticated method. But then what you do is you say, 'Well look, if x hasn't changed much, I'll just replace a of x. I'll just use the previous value, and b will be the previous value here.' So this is like even dumber than calculus because this isn't even a local approximation. This is not a good approximation, but it's often good enough.

So here's an example. Here's a quadratic function. I'm going to rewrite it, many ways to do this, but I'll rewrite it this way as ax + b. So if I quasi-linearize it, I simply take the last version here. Whereas, the Taylor approximation, the correct way to do this is to take this thing out, and these are not the same if you multiply them out. It's quite weird, but they're not the same.

Let's do an example. Our example is going to be a non-convex quadratic program. So we're going to minimize this quadratic over the unit – by the way, it's a famous NP hard problem here. It doesn't matter. It's a famous hard problem. Here P is symmetric but not positive semi-definite. By the way, if P is positive semi-definite, this is a convex problem, and it's completely trivial, and you get the global solution. So it's only interesting if P has a couple of negative iGAn values, one will do the trick.

So we're going to use the following approximation. This is going to be the Taylor expansion, but we truncate, we pull out, we remove the negative part of P because that would contribute a negative curvature component. So here's an example in R20. We run SCP, the sequential convex programming, where the trust region rate is a 0.2. Now the whole, all the action goes down in a box +/- 1. So the range of each x is from -1 to 1, which is a distance of 2. So this thing basically says that in each step, you can only travel 10 percent of your total range.

So in fact, in the worst case, it will take you at least 10 steps just to move across the range here, to move from one corner of the box to the other of the feasible set. You start from 10 different points, and here are the runs. This is the iteration number. That's maybe 20 or something. And you can see that basically by or around 25 steps these things have converged. And you can see indeed they converge to very different things. They converge to different.

So here, if we were to run 10 things, that would be the best one we got. It's around -59 or something. It's nearly -60. That's this thing. Now, we don't know anything about where the optimal value is. This is a lower bound. I'll show you how to derive that in just a second, but that's a lower bound from Lagrange Duality. But in fact, what we can say at this point is that we have absolutely no idea where the optimum value is, except it's in this interval. It's between -59 and whatever this thing is, -66.5. So there's a range in there, and we have absolutely no idea where it is. My guess is it's probably closer to this thing than this one, but you know, it'll be somewhere in there.

You want a better lower bound? Wait until the end of the class, and we'll do branch and bound and we'll show you how to bring the gap to 0 at the cost of computation, and the computation will be a totally different order of magnitude. It won't be 25 convex programs. It will be 3,000 or something like that, but you'll get the answer, and you'll know it. So that's it.

Let me just show you how this lower bound comes up. You know about this. It's just Lagrange Duality. So you write down the box constraints as xi2

Now, if it's positive though, you can just set the gradient equal to 0 and you evaluate it, and you get this expression here. And you can see this expression is, as has to be, this expression is concave because we know G is always concave. We know that by the way, the dual of a non-convex problem, the LaGrange dual is convex. Therefore, again roughly speaking, it's tractable. We can solve it, and when you solve it, you get a lower bound on the original problem. So this is the dual problem, something. And this is easy to solve. You can convert it to a SCP or why bother, let CBX do it for you because that function exists there. So it's literally like one or two lines of CBX to write this down. And then you get a lower bound.

And if you try it on this problem instance, you get this number here. And in a lot of cases actually, when these things are used, you don't even attempt to get a lower value. When you're doing things like sequential convex programming in general, you're doing it in a different context. So you don't have a lot of academics around asking you, 'Is that the absolute best you can do? Can you prove it?' and stuff like that. You're just basically saying, 'Can I get a good trajectory? Can I get a good design? Does the image look good?' or something like that. Or, 'Do I have a good design?' or whatever.'

Okay. That's it. Now you know what sequential convex programming is. But it turns out to actually make a lot of it work, there are a few details. We're going to go over the level 0 details. There's level 1 ones as well and level 2, but we'll just look at the level 0. These are the ones you're going to run into immediately if you do this stuff.

By the way, how many people here are doing project that involve non-convexities or something like that? Okay, so a fair number. These are the highest-level ones that you have to know about immediately. The first is this. You can form this convex approximately problem. So let's go back to what it actually is. There you go. You can form this thing, but what if it's not feasible? Which could happen. Basically, you start with a point that's way off. I mean way, way in the wrong neighborhood, just totally off, and you have a small trust region. So you can't move very far. That means probably this is not going to be feasible.

So right now, if you were to run this, depending on the system, you run, for example CVX, it will simply come back and tell you it's infeasible. Instead of returning you an approximate x, it will return a bunch of nads. It will return you some dual information certifying that the problem you passed in was infeasible. So that's just in case you didn't trust it. It will return you a certificate that's not interesting.

So what you're really going to have to deal with is what you really want to do is you want to make progress. So that's the first thing. Now the other issue is the following: even if this problem is feasible and you step, how do you know you're making progress for the main problem, right. So to evaluate progress, you're going to have to take into account a bunch of these. First of all the objective, that's the true objective not the approximate objective.

So obviously, you want this to go down, and if this point were feasible, we'll see cases where that has to happen. But if this point were feasible, then it's easy how to measure progress, by the objective because that's the semantics of the problem. The semantics of the problem is literally if you have a two feasible points and you want to compare them, the semantics of the problem is if one has a better objective, it's better, end of story. Okay. So that's how you do that.

The problem is what if the approximate problem is infeasible, what if the exact problem is infeasible. You have to have some measure of something it tells you about making progress. So you might say that making progress has something to – you would like these violations to go down. That would be one – that's sort of one measure of making progress. This is like having a Liaponal function. There are lots of other names for these things. Merit function is another name used to describe a scalar valued function that tells you whether or not you're making progress. That's a standard and coherent method.

Now, if you're not making progress, it could be for lots of reasons. It could be that your trust region is too big. And so what's happening is your function is wildly non-linear over the trust region. You're least squares thing is providing some least squares fit, but you're making horrible violations. By the way, when you call the method on the functions that says return an affine approximation, it can also return the errors, right. And if the errors are huge, there's probably no point in forming and solving the convex program. I'm talking about a more sophisticated method. Instead, the collar will actually say, 'Oh, that's too big reduce your trust region by a factor of 2 and try it again,' or something like that.

One reason is that your trust region is too big. So you're making poor approximations. And then what happens, you form a linearized system, and in the linearized system, you have to make progress because it's a convex problem. Unless it just says you're optimal at that point. It will suggest a new point, which will make progress. Any way you form a measure of progress that is convex, it will make progress period. But there's no reason to believe that true non-linear problem is.

Now on the other hand, if the trust region is too small several things happen. One is the approximations are good because you're basically taking little mouse steps in x. You're approximations are quite good. But the progress is slow. You're going to solve a lot of convex programs. And there's one other problem, and this is a bit weird, but it's this. You are now more easily trapped in a local minimum because you're just going to basically slide downhill and arrive at one point.

If you're trust region is big at first and your function has all sorts of wiggles in it, since you're looking down on your function from high up and getting a very crude approximation, you're actually going to do better if you have really tiny small rows, you’re going to get caught in the first local minimum. As you’re figuring, out all of this non-convex optimization is some combination of art, euristics, and other stuff.

Student:[Inaudible] for the row then?

Instructor (Stephen Boyd):Yes, we’re going to get to that. There is, yes. I’ll show you actually what would be the simplest thing that might work. And the truth is none of this works. I mean in the true sense of the word. But if you put quotes around it, then I can tell you what ‘works,’ and that means something comes out and you get something that’s reasonable, close to feasible. Is it optimal? Who knows? But you get a nice picture, a nice trajectory, a nice estimate of some parameters or whatever it is you want.

So a very simple way of constructing a merit function is this. This is the true objective not the approximated one. To the true objective, you add a positive multiple of – this is the violation. That’s the total, true violation, and this is the inequality constraint violation. That’s the equality constraint violation. Something very important here, these are not squared.

They’re not squared, and let me say a little bit about that. They would be squared if the context of what you were doing was conventional optimization where differentiability is a big deal. They would be squared. And someone would say, ‘Well why are you squaring hi, and you day because now it’s differentiable. But if you know about convex optimization, you know that non-differentiability is nothing to be afraid of. That’s why we do this.

But there’s also something else very important about this. It turns out; this is what’s called an exact penalty function. So this is by the way called a penalty method. It’s the opposite of a barrier method. A barrier method forces you to stay in the interior of the feasible set by adding something that goes to 8 as you approach the boundary. So you don’t even dare get near the boundary. Well, you will eventually get near the boundary. Your level of daring is going to depend on how small this is.

In the barrier method, by the way, some people write it here, or we would put it over here with the T and make T get big, but it’s the same thing. That’s the barrier method. A penalty method is the opposite of barrier method. It allows you to wander outside of the feasible set, but it’s going to charge you, and this is the charge. So that’s a penalty. Now, this is what’s called an exact penalty method, and let me tell you what that means. It means the following. It means that – it’s kind of intuitive, and it can be shown that if you increase lambda here, the penalty for being outside the set, you’re going to violate the constraints less and less. It’s obvious, and you can show this.

But here’s the cool part. For some penalties, the following occurs. For lambda bigger than some lambda critical, some finite critical value, the solution, the minimum of this thing, is exactly the solution of the problem. It’s not close. It’s not like, oh, if I charge you $50.00 per violation unit, you get close, and then 80 you get closer, but you’re still violating a little bit. Then I say, ‘Okay, now it’s a thousand dollars per violation unit, and you violate 1e – 3. No, it works like this. It’s $50.00 per violation unit, you violate 80, and I get it up to 200, and your violation is 0, and you have therefore solved exactly the original problem.

Everybody see what I am saying here? By the way, it’s very easy to show this for convex problems. This is non-convex, and in any case, it’s irrelevant because nobody can minimize this exactly. So all of this is a euristic on top of a euristic on top of a euristic, but anyway. So that’s an exact penalty method. By the way, this is related to L1 type things. You’ve seen this there. If you add an L1 penalty, we did homework on it didn’t we? Yeah. So if you have an L1 penalty and you crank up the lambda big enough, it’s not that the x gets small.

You go over a critical value, and the x get small, but for a finite value lambda, it’s 0. It’s just 0 period. So this is the same story. Now, we can’t solve this non-convex problem. No easier or harder than the original problem, so instead, we’ll use sequential convex programming and we’ll simply minimize this problem. Now the cool part about that is you can’t be infeasible here.

I mean assuming the domains of all the Fs and Hs are everywhere, which is not always the case, but roughly speaking, you cannot be infeasible now. You can plug in any old x. You can take a tight trust region method. If you move just a little bit, the hope is that this violation will go down, right. But the point is anything is possible. You can violate constraints, equality and inequality constraints. You just pay for them. That’s what this is.

So that deals with the feasibility thing, and in fact, a lot of people just call this a phase 1 method in fact. So it’s a good method. I should add – I’m going to ask you a couple of questions about it just for fun to see how well you can do street reasoning on convex optimization. So here it is.

When you minimize this that’s a convex problem, and here’s what I want to know. I’ve already told you one fact. If lambda is big enough all of these will be less than or equal to 0, and all of these will be exactly 0. It’s just like L1. So let me ask you this. When lambda is not big enough and you solve these, and some of these are non-zero, what do you expect to see?

So lambda is not big enough to cause all of these to be less than or equal to zero and all of these to be zero. What do you expect to see, just very roughly what?

Student:A small violation for most of the Fi’s but just a few phi hanging around then.

Instructor (Stephen Boyd):That’s a good guess. So a small violation for all of these guys and a few – now why did you say a few here?

Student:L1.

Instructor (Stephen Boyd):Cool, L1 exactly because for example, if these are affine, and this really is, that's really an L1 norm. And so L1 and that part of your brain is next to your sparsity part. I mean the neurons are growing between the sparsity and the L1 thing. That's a very good guess. Actually, I'll tell you exactly what happens. What happens is a whole bunch of these will be 0, and you'll have sparse violations. That comes directly from L1, and you get exactly the same thing here.

So it's actually very cool. You'll have a problem. This is good to know just out of this context just for engineering design. You have a convex problem. You have a whole bunch of constraints. It's infeasible. T hat's irritating. At that point, one option, if you say sorry it's just not feasible. It's not part of your method if it's convex optimizations because there is no feasible solution. Not there's one and you failed to find it.

So if you minimize something like this, what will happen is really cool. If you have, let's say, 150 constraints, 100 of these and 50 of these. Here's what will happen if you're lucky. This thing will come back, and it will say, 'Well, I've got good news and bad news. The bad news is it's not feasible. There is no feasible point. The good news is of your 50 equality constraints, I satisfied 48 of them, and of your 100 inequality constraints, I satisfied 85 of them. Everybody see what I'm saying?

So this is actually a euristic for satisfying as many constraints as you can. By the way, that's just a weird aside. It doesn't really have to do with this particular problem, but it's a good thing to mention.

The question was; how do you update the trust region. Let's look at how this works. So here's what's going to happen. I'm going to solve a convex problem. T hat will suggest and xk + 1. I will then simply evaluate this thing. This might have gone down. These might have gone up. Who knows? The only thing I care about is this phi, and the question is if I went down I made progress, and roughly speaking, I should accept the move. If phi doesn't go down, if it went up, that was not progress, and it could be not progress for many reasons.

It means that your trust region was too big, you had very bad errors, you solved the approximate problem, it wasn't close enough. It gave you bad advice basically. So here's a typical trust region update. By the way, this goes back into the '60s. It's related to various trust region methods, not in this context but the same idea. It's identical to back in the '60s. So what happens is this. You look at the decrease in the convex problem. This you get when you solve the convex problem. And it basically says; if those approximations you handed me were exact, you would decrease phi by some amount delta half. This is always positive here, always.

It could be 0, which means basically that according to the local model you're optimal. There's nowhere to go. Okay? But this is positive. This predicts a decrease in phi. Then you actually simply calculate the actual exact decrease. Now, if your model of phi, if this is very close to that, these two numbers are about the same. So if your approximations are accurate, these two numbers are the same. By the way, that generally says your trust region is too small, and you should be more aggressive. So if in fact your actual objective is some fraction alpha, typically like 10 percent. If you actually get at least 10 percent of the predicted decrease, then you accept this x ~, that's your next point. And you actually crank your trust region up by some success factor.

By the way, this is just one method. There are many others. That's the nice part about euristics. Once you get into euristics, it allows a lot of room for personal expression. So you can make up your own algorithm and it's fun. I guess. Notice that in convex optimization there's not much room for personal expression if you think about it carefully right. You really can't say use my LP code because it's better. 'What do you mean?' 'Oh, you're going to get way better results with mine, way better.' Because you can't because any LP solver that's worth anything gets the global solution. So they're always the same. I mean you can have a second order fights about which one is faster and slower and all that, but that's another story, but you certainly can't talk about quality of solution because they all get the exact global.

In non-convex, you can actually talk about the quality. You can actually stand up and say, 'My algorithm is better than yours.' And it will mean actually that I'll sometimes get better solutions. Here you increase it. Typically, you might increase it 10 percent or something like this. Now if the actual decrease is less than 10 percent of the predicted decrease, then what t says is you better crank your trust regions down. The typical method is divide by =2. This is just a typical thing. As I said, there's lots of room for personal expression, and you can try all sorts of things.

So this is a typical thing to do. By the way, one terrible possibility is that delta is negative. Now if delta is negative, it means this thing – it says I predict you will decrease phi by 0.1, and then when you actually calculate this, it's entirely possible that phi not only did not go down, it went up. That's definitely not progress. But that's the way to do this. This is a trust region update. Like I say, these are things that are easily 40 years old, 50. I mean not in this context, but in the context of other problems.

So now, we're going to look at an example. It's an optimal control problem. Actually, I know we have a couple of people working on some of these. And so it's for a two-link robot, and it's not vertical. It's horizontal because there's no gravity here. That doesn't matter. It's just to kind of show how it is. So there's two links here, and our inputs are going to be a shoulder torque and an elbow torque. So these are the two inputs we're going to apply t this, and we're going to want to move this around from one configuration to another or something like that. We'll get to that.

So here's the dynamics. The details of this not only don't matter, but there's a reasonable probability, maybe 5 percent, that they're wrong because we don't know mechanics, but somebody who knows mechanics can correct us if it's wrong. Anyway, so it's an inertia matrix multiplied by the angular acceleration, and that's equal to the applied torque. That's a two vector. That's towel one and tab two. Here is allegedly the mass matrix. I can tell you this. It has the right physical units. So that part we can certify. It's probably right.

Now the main thing about these is that m of theta and w of theta are horribly non-linear. They involve product of sines and cosines of angles and things like that. So it's not pretty. So this is a non-linear DAE, differential algebraic equation. Of course, to simulate it's nothing to do because we know how to numerically simulate a non-linear DAE. It's nothing. Let's look at the optimal control problem. We're going to minimize sum of the squares of the torques, and we'll start from one position, at rest. That's a two-vector, and we want to go to a target position, again, at rest.

We'll have a limit on the torque that you can apply. So there's a maximum torque you can apply. This is an infinite dimensional problem because tau is a function here, but let's look at some of the parts. That's at least convex functional. These are all cool. This is cool too. So the only part that is un-cool is the dynamics equality constraints, which is non-linear, right. If it were linear, this would be not a problem. So we'll discortize it. We'll take the N times steps, and we'll write the objective as something like approximately that, and we'll write the derivatives as symmetric derivatives.

By the way, once you get the idea of this, you can figure out how to do this in much more sophisticated ways where you use fancier approximations of derivatives and all that, but this doesn't matter. So we'll approximate the angular velocities this way, and we'll get the angular accelerations from this. That's approximately, and those initial conditions correspond to something like this. For two steps, you're at the initial condition, and for the final two steps, you're at the final one. The two-step says that your velocity is 0. That guarantees the velocity being 0, and we'll approximate the dynamics this way.

Now, that's a set of horrible non-linear equations because remember this mass matrix. If this were a fixed matrix and if this were fixed, this would be a set of linear equations. So it's written like that, and the first and lazy thing to do is quasi-linearization. That's kind of the obvious thing to do there. So let's do that. Here's our non-linear optimal control, and by the way, this kind of give you the rough idea of why things like sequential convex programming make sense. That's convex. These are convex. These are all convex. The dynamics are not convex. That's the problem.

We'll use quasi-linearized versions. You'd get much better results if you actually used, I believe – we haven't done it, if you used linearized versions. And if you used trust region versions, you'd probably get even better still, but in any case, we'll just simply – and by the way, the physical meaning of this is very interesting. The physical meaning is this. This is a set of linear equality constraints on theta. But it says it's not quite right. It basically says it's the dynamics, but you're using the mass matrix and the coriolis matrix from where the arm was on the last iteration because that's what this is.

Now, hopefully in the end, it's going to converge. And the new theta is going to be close to the old theta, in which case, you're converging into physics consistency or something like that. That's the idea. We'll initialize with the following. We'll simply take the thetas. We want to go from an initial point to a target point, and we'll just draw this straight line. By the way, there's no reason to believe such a line is achievable. Actually, just with the dynamics, there's no reason to believe it's achievable and certainly not with torques that respect the limits. I mean this is no reason to believe that.

Okay, so this is the idea. So we'll do the numerical example now with a bunch of parameters and things like that, and this says that the arm is start like this, and then with this other one bent that way or something like that. And then, that's going to swing around like this, and the other one will rotate all around. There's a movie of this, but I don't think I'm going to get to it today. Anyway, it's not that exciting unless your eye is very, very good at tracking errors in Newtonian dynamics. There are probably people who could do that. They could look at it and go, 'Ow, ooh, ahh.' And you go, 'What is it?' They go, 'You're totally off.' And then at the end, they could look at it and go, 'Yep, that's the dynamics.'

So the various parameters are; we'll bump up the trust region by 10 percent every time we accept a step. We'll divide by = 2 if we fail. The first trust region is going to be huge. It's going to be +/- 90 degrees on the angles. All the action's going to go down over like +/- p or +/- 180. And then, we take lambda = 2. That was apparently large enough to do the trick. So the first thing you look at is the progress, and remember what this is. This combines both the objective and then a penalty for violating physics. You have a penalty for violating just physics, and in fact, that second term, I'll say what it is in a minute.

But that second term, the equality constraints are in Newton-meters. At every time step, it's two equality constraints on two torques. The residual is in Newton-meters. It's basically how much divine intervention you need to make this trajectory actually happen. It's the torque residual. It's how much unexplained torque there is in it. So this says yes, it's progressing. By the way, this doesn't go to 0 of course because it contains that first term. We'll see how that looks.

So here's the convergence of the actual objective and the torque residuals. So let's see how that works. You start off with an objective of 11. That's great. In a few steps though, it's up to 14. You think that's not progress. The point about this 11 is yes, that's a great objective value. There's only one minor problem. You're not remotely feasible, right. So it's a fantastic objective value. It's just not relevant because you're not feasible.

Presumably what happens, in fact we can even track through this and figure out exactly what happens. So the first two steps our trust region was too big. We were proposed a new thing. We evaluated the progress. We didn't make enough progress. We'll see exactly what happened. We divide by 2 twice. So in fact, it was +/- 90, +/- 45, +/- 22.5. At that point, that trust region was small enough that we actually got a step to decrease things. And what it did was since phi goes down, this thing went way down, and yet, your objective went up. What that strongly suggests is that your torque residual went down, and in fact, it did. This is on a log plot, so you can't see it, but that little bump there is a big decrease in torque residual.

You can see here. This is the torque residual going down like that. So when we finish here after 40 steps, we have something that – surely by the way, this is much smaller than the error we made by discortizing. So there's an argument that we should have stopped somewhere around here, but it's just to show how this works. You can guess that this is the value of the objective that we get in the end. You can see that by 20 steps you actually had a reasonable approximation of this. So this can be interpreted as the following. This is making progress towards feasibility. You've got rough feasibility by about 15 or 20 steps, and this is just keeping feasibility and fine-tuning the objective.

What you see here are pictures of the actual and predicted decrease in phi, the blended objective. And you can see here. On the first step, you predicted this decrease. You predicted a decrease of 50, but in fact, when you actually checked, what happened? Not only did that phi not go down, it actually went up. That's negative. It went up by 5. So you reject it. On the next step, you divided again. The trust region goes from +/- 90 to +/- 45. You try again on +/- 45 here. You try again, and once again, it's a failure. Now you go down to 22.5. You predict a decrease of 50. You got a decrease of 50. That was accepted.

This shows how the trust region goes down. It starts at +/- 90 degrees. You fail once, twice. You're down to 22.5 degrees. Now you go up to 26 degrees. I guess you now fail three times in a row, something like that. I can't follow this. That's about three times in a row maybe. Then you go up, and fail once, and that's the picture of how this goes. And then, this is the final trajectory plan that you end up with. These are not obvious. I don't know of really many other ways that would do this.

So you end up with this torque, and it kind of makes sense. You accelerate one for a while, and then everywhere it's negative over here, that's decelerating. And you accelerate this one to flip it around, and you have a portion at the end where it's decelerating like that. And then, here's the final trajectory. The initial trajectory just went like this. It was a straight line from here up to here, and this one is a straight line from here to here. So I guess theta 2 actually turned out to be pretty close to our crude initial guess.

That's an example. Let me say a little bit about this example. Once you see how to do this and get the framework going, anything you can do with convex programming can now be put in here. And that's the kind of thing you couldn't do by a lot of other methods. You can do robust versions now. You can put all sorts of weird constraints on stuff. You can ask for – since the taus actually appear completely convexly. That's not a word, but let's imagine that it is. In the problem, there are no trust region constraints. So you can put a one norm on the taus instead of a sum of the norm of tau2. In which case, you'd get little weird thruster firings all up and down the trajectory.

That would be something that classical methods wouldn't be able to do for you. This method you could actually do by – this specific problem right here, you could do by various methods involving shooting and differential equation constraint problems. You could do it. It'd be very complicated by the way. This would be up and running like way, way faster than those other methods. But anyway, when you start adding other weird things like L1 objectives and all that kind of stuff, it's all over. All that stuff just goes away, and this stuff will work just fine.

We're going to look at just a couple of other examples of this. They're special cases really or variations on the same idea. Here's one thing that comes up actually much more often than you imagine. It's called difference of convex programming. Although, this has got lots and lots of different names. One is called DCC or something like that but anyway. So here it is. You write your problem this way. You write it as all of these functions. Forget the equality constraints. Just leave that alone. We write it this way. Every function we write is a difference in a convex and a convex program and a function. And you might as by the way, 'Can you express every function as a difference of two convex functions?'

What do you think? The answer is you can. I mean we can probably find five papers on Google that would go into horrible details and make it very complicated, but you can do it, obviously. This is utterly general, but it's not that useful except in some cases. It's only useful in cases where you can actually identify very easily what these are. By the way, this comes up in global optimization too. We'll see that later in the class.

This is weird because it doesn't make any sense. I don't know a better name for this. We can help propagate it by the way if you can think of a name. Difference of convex functions programming, that's closer, but it's not really working. So I don't know what. Anyway, if someone can think of. There's a very obvious convexification of a function that's a difference of convex function. It goes like this. If I ask you to approximate f(x) – g(x) with a convex function. We can do a little thought experiment. If I came up to you and I said, 'Please approximate f by a convex function. What would your approximation be? F. It's convex.

Now suppose I ask you to approximate a concave function? Let's go back and think locally. How do you approximate a concave function by a convex function? You can guess. Linear. Are you sure you couldn't do any better? Well, you could easily show the best. It depends on your measure of best or whatever, but you could write some things down, and surely, people have written papers on this and made a big deal out of it.

Very roughly speaking, depending on how you define best, the best convex approximation of a concave function is an affine function. If you're approximating something that curves down, if you put anything that curves up, you're going the wrong way, and you're going to do worse than just making it flat. You're allowed to do that actually. In the second half of an advanced 300-level class, I'm allowed to say stuff like that.

What you do is simply this. You simply linearize g and you get this thing. This of course is convex because it's convex, and that's constant. Well, this is affine. That's the affine approximation of g. Now here's the cool part. When you linearize a concave function, your linear function is above the concave function at all points. Everybody agree? Your function curve's down. Your approximation is like this. You're a global upper bound.

And what that means is the following. You have replaced g with something that is bigger than g everywhere, and that means that your f hat is actually – have I got this right?

Student:[Inaudible].

Instructor (Stephen Boyd):Minus g, so this means that f hat is bigger than f(x) for all x. That's what it means. Let me interpret what this means. Roughly speaking, a convex function is one where smaller is better in an optimization problem. So it's either an objective or it's a constraint function. If it's an objective, smaller means better. If it's a constraint function and you're positive, it means your closer to your feasibility roughly. If you're negative, it means you're more feasible. Now that doesn't make any sense, but that's good enough. It certainly means you're still feasible.

So this is really cool. This says there are no surprises with the trust region. There's no trust region. If you simply globally minimize, form the function with this, here's what's cool. You just take a full step, follow the logic. There is no trust region. You just remove it entirely. This says that when you optimize with f hat and then actually plug in the true one, your results with the non-convex problem can only be better. All your constraint functions are actually small. They go down, and your objective goes down, right.

So that's how this works. This is much simpler, this method. And this is sometimes called convex-concave procedure. It has been invented by many people independently and periodically, and surely will be invented roughly every 10 years. I have no idea who invented it first, but I think I can guess where. I won't do it. I'll check actually. I do know. It was invented there, Moscow of course. I know that because it's a part of potential methods.

So here's an example out of the book in the chapter on approximations. Here's the problem. You're given a bunch of samples from a distribution. You've subtracted off the mean or something. They come from something with a covariance matrix sigma true. Our job is given a bunch of samples to estimate the covariance matrix. So the negative log likelihood function, which we are going to minimize, is you just work this out. It's right out of the book anyways. It's log debt sigma + trace sigma inverse Y. And Y is the empirical covariance of your samples. So it's that. There's no reason to believe that the Yi's sum 0. They won't.

You want to minimize this function, and we look at the different parts. This is good because that's a convex function of sigma. Unfortunately, that is bad. This is a concave function of sigma. Now, the usual approach in this is to not estimate sigma but to actually estimate sigma inverse. You're welcome to. There's a change of variable. So if you look at sigma inverse as the variable. This is trace times matrix R. This is Tr[RY]. That's linear in R. This is log det R inverse. That's convex. Everybody got this?

If you want to do covariance estimation, there's a very fundamental message. The message is this. You shouldn't be doing covariance estimation, at least from the optimization perspective. You should be doing information fitting. T hat's the inverse of the covariance matrix is the information matrix. That’s what you should be optimizing. At the end, you can invert it.

Now the problem with that is that the only constraints you can handle now are constraint, which is convex in the inverse of the covariance matrix. By the way, that includes a lot of constraints and some really interesting ones by the way. One is conditional independence. If you take samples of the data and you want a base network, you want sigma inverse to be sparse because a zero in an entry of an inverse of a covariance matrix means conditional independence.

So how many people have taken these classes? A couple, okay. How come the rest of you haven’t taken these classes? What’s your excuse? You just got here or something this year?

Student:I did.

Instructor (Stephen Boyd):That’s your excuse? All right, you’re excused. Have you taken it?

Student: No. Instructor:

Why not?

Student: Same excuse came here last fall.

Instructor (Stephen Boyd):Just got here, I see, but you’re going to take this next year.

Student: Yes.

Instructor (Stephen Boyd):Good, right. It’s vase networks. All right. That was all an aside. So this problem you solve by just working with sigma inverse. However, we’re going to do this. I want to say the following. I want to say, ‘No, no, no. Please solve for me this problem, but I’m going to add the following constraint. All entries in the covariance matrix are non-negative.’ Now of course, the diagonals are obviously non-negative. That’s obvious. But this says all of the entries of the variable y are positively correlated.

I pick this just to make the simplest thing possible that is not a convex function in the inverse of the matrix as far as I know. So the inverse of element-wise non-negative covariance matrices, that’s not a convex set. We’re back to that. That means we have to solve this problem. It is not convex, although it’s very cool. That’s convex, this constraint. This term is convex. This is concave. So a concave is negative convex, so it’s difference of convex programming. We can’t use DCP, that’s discipline convex programming. DCFD, that’s not working. Someone has got to figure that out, a name for this.

So this is a difference of convex functions. That's convex minus convex. And so if you linearize the log determinate, you get the trace of the inverse times the difference. This is now constant. This is affine in sigma, and that's convex in sigma, and so we'll minimize that. Here's a numerical example. This is just started from five problems, and these are the iterations, and this is this negative log likelihood. Now, by the way in this problem, I'm willing to make a guess that this is actually the global solution. To be honest, I don't know it and didn't calculate a lower bound or anything like that, but it's just to show how these things work.

Student:[Inaudible].

Instructor (Stephen Boyd):Yes.

Student:[Inaudible] sample of the various [inaudible].

Instructor (Stephen Boyd):It depends. You don't want to be too smart with these methods in initialization. You can start with an immediate initialization, but then you want to rerun it from different ones just to see which regime are you in. Is it always finding the same one? In which case, you can suspect it's the global solution, maybe. Who knows? Or if you get different solutions, you just return the best one you find.

Yes, you could initialize it with Y if you wanted and go from there. How is this one initialized?

Student:[Inaudible].

Instructor (Stephen Boyd):Random, for these 10, there you go. That's how this was done. And it's actually; you want to do that anyway. You don't want to run any of these once. You try them and see what happens, see if you get different ones. I think we'll quit here and then finish this up next time.

[End of audio]

Duration: 78 minutes

ConvexOptimizationII-Lecture12

Instructor (Stephen Boyd):The first is your revised proposals are due tomorrow. So – and we would actually like those maybe stapled – not stapled to – or maybe paper clipped to the original one, so we can see what the original one looked like and so on. As I've said a couple of times, you're more than welcome to grab any of us, me or the TAs, between now and then to sort of just glance over what you have or -something like that. The way you know it's right is that you can explain it to somebody in way under one minute. So if you can explain it in under a minute to another person, not working with you on the project, then that's a sign that you can articulate it completely clearly and all that sort of stuff. Okay. So those are due tomorrow.

And then the other thing is that a week after that will be these midterm progress reports. And then that's basically the same thing. It's – takes your three page thing, but now it's four pages, or maybe even five. So – because it's now got – this one should be your statement of what you're looking at, what the real – you know, what the problem is, and maybe something about the approach. That's what we should be seeing tomorrow. And by the week after, especially because these days it's not a huge deal to actually implement any of these things. It takes, like, 15 minutes to write scripts for a lot of these things. Then by next Friday, we'd expect to see at least some results about how these things – I know that some people are way ahead on that. And there's actually two different – some people are way ahead on that, and maybe way behind on the writing, and vice-versa, and stuff like that. So we'll make sure to keep those sort of on track. Oh, the other thing is I guess we're confirming May 30, which is the Friday – the week before the last week of classes – as the due date for the project reports. So – of course, by then we'll probably have read your project reports about 45 times. So it'll maybe just be a formality at that point, which is fine. We don't mind doing it. So as long as we see serious progress each time we take a look at it. So that's gonna – that'll be that Friday. And I think what we're gonna do is we're actually gonna have a – there'll be poster session, and it's gonna be maybe an afternoon or evening, the week of the last week of classes. So that's, like, June 2 or something like that. But that's – it's on the website. And so that's what we're gonna do, and it'll maybe be some evening or afternoon. And the posters – I mean you'll hear more about this, but it's totally obvious what it is. It's your standard 12 pages and 12 slides. Nothing fancy. This will just be – you'll print out 12 pieces of paper. We will have to figure out where on earth we're gonna get a bunch of poster boards or something like that, but we'll – seems like we can figure that out. So we'll do that. And it'll be interesting. That'll be for everybody to see everybody else's project. If a project is well – if it's already – if it's well written, it's shockingly easy to write a talk. So – in fact, you'll find out you've already written a talk because the sections of your report, or whatever it is – I guess it's the report at the time, will simply be a talk, except it will be in 12 point, and you just blow it up to the bigger one. You can use our – we have some templates for talks. So.

Let's see, a couple of other things. Oh, you should watch for email, and maybe the web site, because we may tape ahead next -Tuesday's lecture tomorrow some time. So – and we would. You'll hear about that. Okay. So, I can't think of any other administrative things. I guess we posted a new homework, but I guess most of you know that. Any questions about last time? Because if not, what we're gonna do is we're gonna finish up this topic of Heuristics based on convex optimization for solving the non-convex problem. So that's the first thing – we're just gonna finish that up, and then we'll move on to another little coherent section of the course, which is basically how to solve absolutely huge problems. So for some of you – I mean if you've already chosen a field, and it's too late, this will be relevant. -And if you make the mistake of -choosing a field where you have to solve huge problems then this will be – these will be the – it would be two or three lectures – these will be for you. For those of you who haven't chosen a field yet, this will give you – it'll let you – this will be encouragement to not choose a field where vast and huge problems are requires. So that's my – that will be my recommendation. Okay. So let's do difference of convex programming. Let's finish up this step. I think we looked at this last time. So this is a topic or method where you write all the objective and inequality constraint functions. You can do with equality constraints functions, too, but I'm not gonna go into it. And what you do is you write them as differences of convex functions. All right, so convex plus a concave function. And then the obvious linearization – convexification, I should say, is this. You linearize the concave functions. So only the concave portion – that's minus a convex one – is – you linearize that. And then this is roughly speaking, based on the idea – I mean you can make this precise, but it doesn't need to be – is the that the best affine approximation of a – the best convex approximation of an affine function is an affine function. All right. So – okay, so you do this and the result – this is now convex because the convex part you left exactly as it was, and the only thing you've done that's a constant is linearized – that's this thing – the concave portion. So – by the way, in a – this – the way to do this is, for example, in – it's actually pretty straightforward. -The way you would do this for a quadratic, you can work out how that is. If you have a quadratic that's neither convex nor concave, that's split eigenvalues. You split into the positive part, and that's gonna be this thing. And then the negative part, you simply linearize, but you drop – you will drop the quadratic term. And then that gets linearizing, it's just dropping the quadratic term. That's only for the concave part. And actually, you'll find out it's quite interesting. If you have a function that's convex, that this -difference of convex, concave, it'll actually have some negative curvature. And basically, what you're doing is your just flattening the negative, but you've retained, exactly, all the positive curvature -portions. Okay. So an example we'll look at is from the book. It's estimation of a covariance matrix from samples. So you form the sample mean, and the negative log likelihood function is log debt sigma plus trace sigma inverse one.

Now, I mentioned this last time that the general approach now is to do the following: is not to estimate sigma, but sigma inverse, which is – that's a one-to-one transformation, so you're perfectly welcome to do that. In which case, this function is convex in sigma inverse because that's linear, and that's convex because it's logged at sigma inverse. Well, it's logged at sigma inverse inverse, so – which is convex. So that's the standard method. Now, that works, provided the constraints you want to impose on sigma are convex in the inverse of the function – sorry, inverse of the covariance matrix. Actually, there's a lot of interesting constraints that are convex in the inverse of a covariance matrix – many. But some are not, and actually, we just want to illustrate DCC, or whatever this is, a convex, concave procedure, something like that. So we just picked on that's not. So suppose I told you that all the variables are non-negatively correlated. There are no negative correlations. So that would – that's this. And this is not preserved under inversion. So the set of matrices whose inverses are non-negative is not convex. Okay. So that leads us to this problem here, and we have a convex part and we have a concave part. And we linearize the – we're gonna linearize the concave part. That's this. I mean this doesn't matter because it's a constant. And we're gonna preserve this convex part. So – and the results here, I think we looked at this last time. I just wanted to make sure I went over this – well, is this. And my guess is that this is actually globally optimal. And the reason is actually there's – well, there's even an exercise in the book that goes over this. It turns out that in some cases, this problem is actually convex insigma, depending on the final estimated sigma versus the covariance matrix. So that's this. It's just to show you how it works. Okay. We'll look at alternating convex optimization. That's our last topic. And we'll look at a couple of examples. So this is another idea that comes up. Lots of times, it's discovered periodically, will be discovered periodically in the future by many people in many different – maybe one of you will discover this again sometime in ten years, or even less. That's fine. I mean, actually, there's nothing wrong with discovering it. The only thing wrong with discovering it is imagining that, in fact, you're the first to discover it. I mean that's the only actual problem here, which there are people who do. So, okay, all right, so this works like this. You partition – these aren't actually started – not partitions. I have index subsets here. And you assume that for each of these index subsets, the problem is convex in a subset of variables.

So that's sort of the – that's the picture. Okay? And in fact, we can -even, if you like, look at an example just for fun because that's the only way this really makes any sense. I wonder if you can raise this screen up to about here. That'd be great. Okay. That's good, - right there. So let's take a look at that, and actually, let's look at something extremely simple. Things don't come simpler than that. And let's imagine the variables are actually, for some reason, A, X, and B. So that's three groups of variables. And now, I want to know is that convex in A? What I mean by that, of course, is that X and B are fixed. Is that convex in A? Of course it is. It's affine. It's a norm of an affine function. So it's convex in A. Is it convex in X? Obviously. Is it convex in B? Of course it is. Okay, now, let's do pairs. Is it convex in X and B?

Student: Yes.

Instructor (Stephen Boyd):Yeah. Is it convex in A and B? Obviously, right? And the final one, is it convex in A and X? And the answer's no. Okay? So this would be a case where there's all sort of index sets. So the index might be something like this. And you might as well get maximal ones, right? So AB – I guess that's about it, right? If you're gonna do that. And you could have XB. Maybe these are the maximal – sets, something like that. Those are the maximal sets, index sets over which the problem is convex in each set. Okay. Now, the method now goes like this. I mean it's really kind of dumb. For each of these index sets, you – it's convex optimization to solve that problem. So you do that, and then you kind of go back and forth and so on and so forth. And hope that this works. So actually, let's even look at an example. Suppose I ask you to solve this problem. So here would be the problem. I should remember those sets, actually, AB and, I think XB. So, in this case, we're asked to minimize this – note my period – okay, subject to that X in some – X – this is convex. Right. A and some script A and B and some script B, okay? These are all convex sets here. So obviously a non-convex problem, that's clear. It's non-convex. Why? Because of the AX there; actually lots of problems come up this why, incidentally. In fact, there's even projects that are literally, directly this blindly convolution. I think – is that yours? Yeah. It's yours. So there's plenty of things that come up looking like this. And so this method would look like this. It would alternate, and it would actually say first, fix X to the current value. And optimize over A and B. And that's a convex optimization problem. That would update A and B. Then is says fix A and B, and optimize over X and B. So notice that B is being optimized, actually, every step. I mean this is not a big deal, but it's just to illustrate kind of how this works. So that's the picture. Let's see. What can you say about this? Nothing. I mean that holds for this entire lecture.

But, I don't know. You can – I mean you can say trivial, stupid things. But after a while they get kind of – you know, you realize, like, why bother? I mean, you know, if there's theorists – I don't know. It's – look, obviously when you do this, every time you do it, the objective goes down. So amazing, isn't it? The objective actually – it's a number that goes down. Therefore, it converges – I mean, if it's bounded below. So – I don't know. And then, so what? So – okay. I mean I don't know, that's better than nothing, or whatever. But it's so obvious that I don't even see the point of stating it. Okay. So a special case is actually when you partition the problem into two variables. And you just alternate. But there is lots of variations on this. That's the nice part is because it's an algorithm that often works, but need not. It means, actually, that for this whole lecture on non-convex optimization is that there's lots of room for personal expression, which is not the case, basically, for convex optimization because, you know, a method that solves a convex problem, like – doesn't badly is just called a non-method, right? And any two methods get – I mean if they disagree on the answer, one of them is – I mean and one of them is a not acceptable method, right? Well, for a non-convex optimization, who knows? It's very nice, you know? Maybe your method works better one problem instance; mine on another. I mean, so it's very nice. You get room for personal expression. I can mention some of the variations that you can do for this. One is actually to not optimize all the way, but to take a fractional step in between. So that's an option. It doesn't sound like it makes sense, but actually that can actually make these work better and so on and so forth. But just remember this is – that's alchemy. That's just like – who – it's just go ahead and try it and see what happens on your problem. So that would be – in other words, you would optimize, for example, over A and B, and not step A and B all the way there. But step them half – some fraction – halfway. Okay. All right. So this is alternating convex optimization. Here's a recent and interesting application of this is non-negative matrix factorization. If you don't know about this, you probably should. Although the people popularizing it are making way too big a deal out of it. But that's okay; they're welcome to do that because it is really cool. Here it is. It's sort of like – you might call it a signed, singular value decomposition. So basically, I give you a matrix. I want to explain it, somehow, as a low rank matrix, as an outer product, something like this. And you should imagine that K is small, here. So K – you might want to approximate matrix A as an outer product of a – I guess in this case, a M by five, and then five by M. So you'd have sort of, like, five factors or something.

Now, without this, this is completely solved. It's just the SVD. You take the SVD and you truncate the SVD expansion, and you get the answer. Let's see – oh, by the way, it's EE even – I mean there's lots of – it's not unique here, and that's completely obvious because I could – I can put a positive matrix – you know, K by K inside here, and I get the same – I get a different factorization that yields exactly the same objective value. So it's not unique. But that doesn't really matter. Let's see, a couple of cases we can solve. Oh, you do K equals one. So we can – you can actually approximate a matrix in Frobenius norm, well, any norm. It doesn't matter, a Frobenius norm or a spectral norm in – with a outer product, which is restricted to be positive. And that goes in two steps. You first take the positive part of A. That's the first thing you do. And the next thing you do is you take the singular value decomposition of that. Now, it turns out there's something called Perron-Frobenius Theory that says that the singular values of a positive – singular vectors of a positive matrix, the first ones associated with sigma one, which is what would come up in the rank one approximation, those can be chosen to be positive. Okay? So that would – that's one case where you can do it.

Okay, now on the other hand – I mean come on. It's just this thing. You look at this and you realize, "Wow. It's certainly convex. And X of Y's fixed. And it's convex in Y if X is fixed." And so you can simply alternate over these. And you can say a few things. I think in one of these variables, it's actually separable. I forget which one. I don't think it matters much. Maybe in the columns of Y or – I – it doesn't matter. It's separable. Actually, the separable is not that exciting unless you're gonna do – unless you're gonna run on many machines. Unless you're gonna do an MPI or something like that. Because, as we've discussed several times, if a problem is separable, it just means the computation time grows linearly with the problem size. But that's the case even if there's light coupling by these sparse methods. So, okay, this is non-negative matrix factorization. And here is an example with a 50 by 50 matrix. And we want to show it – we want to approximate it as a rank five matrix. And then, here it is starting from, say, five starting points. And you can see it's going down like this, and will converge. It need not converge to the – in fact it's very – it's unlikely to converge even to the same – it might converge, in this case, in the same one, but there's no guarantee and all that sort of stuff. So all of these methods – these non-convex methods, they're extremely useful. So the fact that no one – you know, if you're doing the Netflix challenge or something like that, and this method works, I don't think you're gonna sit around and complain and lose sleep over the fact that you're using a method that you can't prove always gets the global minimum. So this lots – you have to remember there's lots of cases where this stuff is fun. It is actually – I mean it works perfectly well. If you don't get the optimal trajectory for your robot, as long as you reliably get a really good one, that's fine. There's other cases where you can be much less – you can't be as casual about not finding the global solution for something. But this one – yeah?

Student: In this problem, did you create A to be the product of two positive matrices? [Crosstalk]

Instructor (Stephen Boyd):No, we didn’t. I think it's just random. All the source codes are online. You can look at it.

Student: If you did this, would you get the answer, usually? Or –

Instructor (Stephen Boyd):I think you said it just right. Yeah. So yeah, the obvious thing to do here is to make A – you know, generate A as XY and see if you get back XY. By the way, you won't get XY, right? Because there's a – what do you call it? It's not unique. But you'll get something – you should get – they should have the same ranges or whatever. They should be unique Modula of this transformation. Sorry, you should get the right X and Y back. Yeah. I think what happens in that case is you often get back. So I think you said it exactly right. You usually get back what you were looking for. So it's heuristic. You do not have to get it back. For one thing, it depends on where you start from. So – yeah, but these things work, like, pretty well, often. Okay. So that finishes up this. I should say that this was sort of – it was out of order for the class. It was out of order because it – that was just a whole lecture on non-convex methods kind of stuck in the middle of the class. It kind of logically goes at the end, but I thought we'd switch it around for the benefit of those doing projects that are non – involve non-convex problems. Yeah?

Student:[Inaudible] SVD solves like a non-convex problem.

Instructor (Stephen Boyd):It does. That's right.

Student: So are there extensions of SVD that are guaranteed to find the [inaudible] –

Instructor (Stephen Boyd):Yes.

Student:– somewhat similar non-convex problems.

Instructor (Stephen Boyd):Good. That's a great question. So a good question is, "What are the problems that aren't convex that you can actually solve?" And now we're back, by the way, now that that's lecture's over, we're back solve now, once again, the – we've just went over the right bracket and the scope. Solve now means solve globally. It doesn't mean this kind of like maybe, possibly get a local solution sometimes, which is what it meant in the last lecture. So we're back to the global definition because that block just finished. And now let's answer your question. So the question is – so clearly there are non-convex problems we can solve. One is things involving like, what's the best rank for the approximation of this matrix? That's – we do that by S – truncated SVD. It's – that is plenty non-convex. And it actually – so the question is what are the others? And there's a handful of others. So there's some – we can solve any problem that involves two quadratics, convex or not. That's just – it's actually not obvious. And then you can involve some weird things, like things involving – some things involving quartacs. But it depends on the dimensions. And these are just by sort of accidents of algebraic geometry. But we're not gonna talk about it. But the point is – I guess my opinion is there's just a handful of problems that are non-convex that you can actually solve. And I mean in the strong form. Solve as in get the solution. Right? So if that's what you mean by solve, there's just a handful. SVD is one, and you're gonna find plenty of others. But –

Student: Can I ask you a question about quadratics?

Instructor (Stephen Boyd):Yeah.

Student: So if you have a convex, concave problem –

Instructor (Stephen Boyd):Um hm.

Student: Why can't you maybe, like, approximate one of them with one quadratic, approximate the other one with another quadratic, and then inherently use that to solve –

Instructor (Stephen Boyd):That doesn't solve the problem. What you can do is you can minimize any quadratic function, convex or not, subject to any single quadratic inequality, convex or not. Okay? You – by the way, you know some examples of that, like, how do you minimize X transpose AX subject to X transpose X equals one? Answer? Minimum eigan vector. Okay, and that's a non-convex problem, if A is not positive semi-definite. If A is positive semi-definite, well – well, actually with the constraint there, X transpose X equals one, it's non-convex there. So yeah. So there's a handful of these, but not a whole lot. But they're good to know about. I mean, so – that's actually in the appendix of the book, if you want to read that. It's an advanced topic, and that also is something, which is – that's actually something deep, unlike difference of convex programming, which is basically just an obvious kind of trick. That's actually deep, and you can actually say – I mean that's a strong statement. And yeah, people periodically – about once every ten years – rediscover that in different fields. It's got – and so it's got lots of different names like Parrot's Theorem – I don't – you – look in the appendix because we listed some of the things there. Okay But if you have two quadratics, you can't do it. Although there are some weird, isolated cases where you can solve it with two quadratic constraints. So this is – it's weird. And then there's weird things where there's, like, quartacs. And so there's just a handful of random things that people who study these things would know. And by the way, in almost all of those cases, in a – not in the eigenvalue case, but in some of these cases, it turns out that these non-convex problems you can solve are basically because you form a convex relaxation, solve that. And then you prove separately that the relaxation actually can always be rounded to – it's always tied or something like that. So that's how those work. And those always involve real stuff. If you look in that appendix, it's not simple. So – okay, so that finishes up that. Like I said, we had a right bracket, so that means we're back to convex optimization again. And we're gonna start another chunk of the class. It's another coherent chunk of the class. And it's basically on how to solve extremely large problems. So basically, as big as you want, if you can store the data. I mean basically, we can solve the problem with a memory that’s basically on the order of the size of the data, number one. And with a computation that's completely linear. Right? So when you get to problems that are a million variables, ten million and more, N squared two doesn't do the trick, basically, either in storage or anything else. So – I mean not even remotely close to that. You can go – you can handle an N to the one point one, or something like that. Or one point two. Beyond that, forget it. So we're gonna look at these methods. They're actually extremely good and extremely cool. And I think just any educated person should know these things anyway, even if you've made the wise choice to go into a field where the number of variables is and will remain for the foreseeable future, under a thousand or ten thousand. But if you're in a field where things are gonna grow, this is gonna be part of your future. So, okay. So [inaudible] radiant method – by the way, how many people have seen this somewhere? Oh good. Okay. Where? Some ICMA thing? Or – okay, so you'll know everything.

Student:CS 205.

Instructor (Stephen Boyd):CS 205. Oh, okay. Great. That was –

Student: Mathematical methods for Vision, Robotics, and [inaudible].

Instructor (Stephen Boyd):Oh, okay. Good. Okay, good. Great. So a bunch of you have seen it. You probably have seen more of it than I have. But we'll – so for you it'll – you can just sit back and, I don't know, relax or something. So for the rest of you, it's great stuff to see. So first, let me start with just sort of the really big picture about solving linear equations. Now, I'm sure all of you know the following. I've said it enough time, but it's important enough. I'll say it again. If you profile, most – certainly, most convex optimization methods, if you profile them and ask if it's – while it's running, what is it doing? It's solving linear equations. That's it. And there's a bunch of other stuff, but it just doesn't even show up in profiling, like, you know, the little line searches and evaluating logs. Zero. What really matters is computing the steps. And that's solving linear equations. So, okay. So – by the way, it's true for a lot of problems. It's just solving linear equations. So let's talk about, at the very big picture, how do you solve AX equals B? Okay. So the – it's kind of a crude classification. It's actually wrong because there's – the boundaries between them are gray. But, in fact, it's helpful to classify these into three huge classes. And here they are. One is dense direct. These are factor-solved methods. We covered this in 364 A. You've probably seen it other places. This – I mean the software involved there is almost universally BLAS in LAPACK. That's – I mean if you use metlab, that's what you're using. Actually if you use anything reasonable, that's what you're using. So these are factor-solved methods. You do – you factorize A into a product of matrices, each of which is easily solved. And then you do a back and forward substitution or something like that. Now, the – here, the high-level attributes of these methods. You can take an entire class on just this, right? No problem. It's like CS 237 or something like that. But – and now we're going through all the horrible details of how do you do the factorizations? What are the round off error implications and all that kind of stuff? But at the highest level, here's what you need to know about these methods. The run time actually depends only on size. That's not completely true, except for positive definite systems, in which case it is true. So for a indefinite systems, you do dynamic pivoting, which means that you actually – there actually are conditionals in the code. You look and you look for a new thing, and you reallocate – and unless you've allocated memory correctly ahead of time and stuff – you know, if you – well, no, I guess in this case that wouldn't happen. Sorry, scratch that. That's coming up.

So there are conditionals, and that means that actually solving AX equals B for a general matrix A actually is not independent of the date. If I give you one A or another A, it can actually take slightly different amounts of time. Actually, it's absolutely second order, so you won't notice it. But it's not deterministic. If A's is positive definite, it is absolutely deterministic because in Cholesky factorization – you do not pivot in Cholesky factorization. And actually a compiler will unroll the loops, and it's just – you know, it's whatever it is, N cubed over three operations, all pipeline, and it just shoots right through. And there's no – it can never be different. It will be repeatable down to the microsecond or lower. Okay? So that's it. Well, there's a few other that happen. Of course, it can fail. I mean there – it can throw an exception if you pass it a singular A. So – but roughly speaking, other than that, it's just – it’s deterministic. It has nothing to do with the data, except of course when to the boundary where A is singular, it doesn't have anything to do the structured of A. If you toss in a toplits A or a sparse A or bandit A, this thing doesn't care. Okay. And this will work well for N up to a few thousand, I mean immediately. And you can do the arithmetic to find out. It has to do with how much will fit into your RAM and stuff like that. So this will work up to a couple thousand. If you want to use a bunch of your friends' machines and use MPI or something, you can probably go bigger than that. But that's – you know, you can just – I mean it will actually – this will scale up to much bigger problems, but not on a laptop or a little machine or something like that. Okay. So the advantage of these is that it's very nice. It's just technology. This is just A backslash B at the highest level, just done. Right? Extremely reliable, you know. When it fails, it fails generally because you gave it a stupid problem to solve. That is to say A was near singular for some practical purpose.

Okay. Now, this is – so this is the first level. The next huge group is called sparse direct. So these work like – direct means you still are factoring A, but this time you're taking into account sparsity in A. And these – I mean sometimes there are well known name structures, like A could be banded. That's a case where it's named. Or it could be a sparsity – just a sparsity pattern in the sense that there's just lots and lots of zeros all around and so on. Okay? So these are – actually those are sort of different cases, but they're – I would call them both sparse. I'd put them in this part. Now here, the run time is going to depend on the size. Of course, that's always the case, the sparsity pattern. And it's almost independent of the data. It's actually a little bit less independent of the data than it is here. And the reason is in the general, unless it's a Cholesky factorization, you're actually doing dynamic pivoting. And in this case, dynamic pivoting has fill in effects. So you may even end up – you know, if you end up calling Mallock or something in the middle of one of these things, you're gonna be sorry. So, in fact, the way these typically work is you'll actually allocate – before you even go in, you'll allocate a block of memory hopefully big enough to contain all the fill-in that is gonna happen because as your dynamically going through it, you're generating fill-in because you didn’t like the pivot that you would have taken, or something like that. Okay? So that's what this is. And these will – it depends on the sparsity pattern. This'll take you up ten to the – I mean generally, ten to the four pretty easily. Ten to the five if it's special, like if it's banded or block banded or has some arrow structure or something like that, you can easily take this up to a million. I mean if it's – if the sparsity pattern is – well, let's say it's banded. You can go to a million like that. So my laptop will easily go up to a million. No problem, none. You can just work out. You just figure out how much DRAM it takes. It's just totally straightforward. By the way, this is described, at least in a very high level – more than I'm gonna say about it here. This is in the appendix of the book on convex optimization. Of course, there's entire books on these subjects, too. Many – and a lot of them are really good. So it depends if you want that much detail. But it's all there.

Now, the interesting thing about these is I – you can argue that these are completely non-heuristic. They always work. You know, if you type in a random matrix, they'll dense. And before A backslash B, I'll be able to tell you how long it's gonna take. I mean if I know a little bit about your machine. And I will not be wrong. I absolutely will not be wrong because I know what it is ahead of time. We can guarantee it, all that kind of stuff. I can embed it in a real time code, and we just know how long it's gonna take. End of story. These actually are semi-heuristic. And so you have to be a little bit careful about this. What that means is the following – after a while you get used to these – to sparsity being handled. Then we did – there is some heuristic in the sparsity stuff. And the heuristic comes into the ordering. So how you pick the permutation to do your stuff. That's a heuristic. And there's actually – that's actually a whole field which is very interesting, but there is a heuristic, and it's this. So when you type in your problem, and you have 100,000 variables in AX equals B. And you type A backslash B, I mean it's as if you're doing mad lab. But remember, they didn't write any of the numerics, right? It's all doing stuff that's – it's written in C and stuff like that, in that case, probably a UMF packer. I'm not quite sure, but that's – I guess that's what it's doing. What will happen is this. You may – there is a moment where you – of tension. And it's either gonna take like two seconds or twelve hours, or something like that. And it's very likely to take two seconds, right? But that just depends on your sparsity patterning and the ordering method you're using. So, okay, okay, and now, we get to the third – oh and by the way, these are not – I'll say it's a little bit about I think the boundaries are gray. Any sparse direct method does this. When it gets near the end, and it's got a little block at the end it's working on, and it's all dense, it – usually when you get down to like, 1,000 by 1,000, it just says screw it, I'm going dense. Even if there's lots of zeros in it, it just goes LAPACK at that point, just BLAS LAPACK. So that's – so a real sparse matrix solver, when it gets – if the fill-in gets big enough it just says done. LA BLAS, LA PACK and does it. So that's why these are – the boundaries not exact here. Now, we get to third class, and it's the one we're gonna study now. It's interative methods, also called indirect. If you do any – this is basically how all scientific computing does, so large-scale. Your example of that, it's PDE solvers. It's anything large-scale. Typically anything with a million variables and more, you're talking interative. I mean, unless it's very structured, like banded or something; you're taking interative.

Now, all of – like I said, all of scientific computing and so on. Now, these methods – and notice that we're losing reliability every time we jump up one. The first one is like it just always works. I don't even have to hear your data, and I can predict exactly how many milliseconds it's gonna work. It is 100 percent reliable, unless you throw in something –

stupid data at it. Then you get to sparse, I can say, "It might – it probably is gonna work. You know, it depends on your data. It depends on the matrices. It also depends on the ability of your – of the heuristic for choosing the ordering, whether that's gonna actually give you a good low fill-in factorization." But after that, it's not – it's probably – after that it's gonna work, once you know the fill-in story. Here, the run time actually depends on the data. So – and the required accuracy. So the other two methods are basically producing very accurate solutions. And it's a real issue, which I'm not covering. But that's – here, this is – these things are tough. These require tuning, preconditioning. That's something we'll talk about. And I'll say a little bit about that. So these are less reliable. They need babysitting. These are not just backslash type things, where – you don't embed these in there. You have to tune it. Now on the other – and you'd say, well, that's a pity. Of course it's a pity. I mean imagine what a pain in the ass it would be if every time you type A backslash B, you actually had to tune or tweak, like, six parameters to make it work. So that's – welcome to the world of super large-scale problems. It's tweaking. You know, that kind of stuff. Preconditioning, we'll get into that. So you could say it many ways. Lots of – there's lots of opportunity personal expression here. A lot of this is not fun, too. And therefore, it often falls to graduate students to do, just to warn you. Just to – I don't – it doesn't mean anything to me, but trust me. This is the kind of thing – ideal for graduate students to do. Okay. And you can complain about this all you like, but the point is, if you're solving a problem with a million or ten million variables, and it's actually – something's actually coming out, you hardly have the right to say, "God. It's a pain in the ass that that poor grad student had to work for three months on the pre-conditioner. And the solutions aren't that accurate." You'd say, "Hey, look. You know, it's your – you were the one who decided to solve a ten million variable problem. Don't – you know, you can't –." There's not much you can say after that. Okay, so this is the third method.

By the way, the boundaries between them are actually very – they're a bit gray. So for example, people will often use indirect – a few steps of an indirect method, or iterative method. Actually, after one of these, that's called interative refinement. People will also use a pre-conditioner. We'll get to that. Is a pre-conditioner, which is, in fact, based on an incomplete factorization or something like that. And then – so at that point who knows what these things are. So – but it's useful to think of these three huge classes of methods for solving linear equations. That's very useful. Okay, so now we'll get into one, most famous one. Well, they're all – a lot of them are the same. After a while, you get it. So we're gonna look at symmetric positive definite linear systems because actually, mostly that's what we're interested in. We solve either Newton equations, Newton systems, which is H inverse G minus H inverse G. Or even if we're solving with equality constraints, we have a big – an H, A transpose A zero, a big KK key system, which is indefinite. But if you do block elimination, you end up solving something that looks like A, H inverse A, transpose, or something like that. So a lot of it comes down to solving this. Okay, so this comes up, for example, in – well, in Newton. A Newton step is that. It's just – it's least squares equations, regularized least squares. If you minimize just a convex quadratic function – and of course this is solving – this is also solving an elliptic PDE, like a bauson equation, if you discretize. So that's what you end up doing. So, okay, as another example, it's sort of disanalysis of a resistor circuit. So this comes up also in circuit stuff. So here that you have a conductance matrix, and I is a given source current.

And you want to calculate all the known potentials, or whatever. By the way, same thing comes up in analysis of reversible mark-off chains. That would be another one that I didn't put here, but it would be the same thing. Actually, it's related to this. So if I give you a reversible mark-off chain, or something like that. And I say, "If I start the thing here at this state, and I want to know what's the probability that it hits this state before that one, or something." You're basically solving a problem like that as well. So – or if you want to calculate, like, hitting times and all this, it's the same sort of thing. Okay, so now we get to CG. Actually, there's a bit of confusion over the method. It's sometimes called conjugate gradient, and sometimes conjugate gradients, with the "s" there. And it's kind of – I don't know. I may switch between them.

And then, I think it also depends if your first exposure to this is in a – if it's – I think one is the British style, and one's the non-British style. But I – anyway, but just to warn you, you're gonna see "s" – sometimes "s," sometimes not. Okay, so this is – it was, as far as anyone knows, it was first proposed in 1952, by Hestenes and Stiefel. And it was proposed as a direct method. And now here's the – instead of getting into the details of the method – in fact we won't get to the details of the method until well – actually the details of the method are actually irrelevant. That's probably not how you guys learned it. But we'll get to that later. There's actually many actual methods that will generate the same sequence of points. So – but we'll get to that later. Here's what it does. Here's – high-level attributes are this. It solves AX equals B, where that's symmetric positive definite. Now, in theory, if you were to do exact arithmetic, it takes N iterations. That's the dimension. Each iteration requires a few inner products in RN; that's fine. And it requires a matrix vector multiply. You have to make it – you have to implement a method that given Z, will compute AZ. So that's what you have to do. That's the – that's basically what you have to do. Okay, now, if you work out that arithmetic of A is dense, then no problem. You call A Z, and if A is dense, you're multiplying N-by-N matrix by vector. That's N squared flops. And congratulations, you run CG. It takes N steps. Each step is order N squared, and you're back to N cubed. Actually, what you have now is an N – is an order N cubed solver that is much less reliable than the standard ones, and actually much less slower because you're using BLAS level one or two. I guess that's BLAS level two because you're using matrix vector multiply, whereas the real method for solving would use BLAS level three and things like that, which would be block, block. And it would use – that would be optimal. Okay, so there's actually no reason to use CG – a CG-type method. I guess, to tell you the truth, CG is really a particular method. And honestly, this really should be called Krumlauf subspace methods. I should even change this slide. I feel so weird because it's really a specific method, but they're all kind of lumped together. Okay, all right, now, if ever matrix vector multiply is cheaper than N squared, you can do better. Okay, so here's an example. If A is sparse, A's multiplying Z by AZ is fast because you only multiply by the non-zeros.

Now, if it's sparse, and not absolutely huge, you're probably better off using a sparse method, like a sparse direct method, maybe, okay? But let's see, someone give me an example of a matrix, N-by-N, you can multiply a vector by fast – fast means less than N squared – but which is not – but which is full. Not sparse. What's an example? Again, all of you have seen these.

Student:[Inaudible].

Student:A counter-product matrix? Like if it's –

Instructor (Stephen Boyd):Perfect. Actually, that's perfect. Yeah. So a matrix that's, like, diagonal plus low rank. Yeah. Diagonal plus low rank. Oh, I mean you have to know it – represent it that way.

Student: Yeah.

Instructor (Stephen Boyd):Perfect example, diagonal plus low rank is clearly full – I mean generally it's gonna be dense, and yet you can multiply – if you know that method, you can multiply real fast. Anything else? This is classic, undergrad EE.

Student:FFG?

Instructor (Stephen Boyd):FFG. So I think – and the right way to say it – let's say it right. A would be the DFT matrix. It's N-by-N. And you would carry out the multiplication using the FFT algorithm. So, and that would be N law again, as opposed to N squared. So that's gonna – that's an example. There's a whole bunch of others. We'll probably mention some of them. Now, we get to the bad news about CG, and this was recognized already in the '50s. It works really badly because of – with round off error. So some methods the round off error doesn't really matter, or you know, it has an effect but the errors don't build up as you go. Others, it's just a disaster. This is just, almost universally, a disaster. I mean, except for the baby examples I'll show with, like, ten variables where CG would be highly, highly, highly, highly inappropriate to use. So, otherwise, it just basically – it's almost universally, it works terribly. So that's the – out of the box, we're gonna see some tricks that will make it work reasonably. Okay, but here's a very important attribute of this method. And it's this. Depending on A, B, you know, some luck, the graduate student that put it together, put it all together – well, no, that's – then what happens is kind of cool. In a very small number of iterations, you can get a rough approximation of the solution quickly. That's a very different thing. If you go back – let's do – let's solve a circuit. I have a circuit with a million nodes. I pump in some currents. This is one step in running spice, let's say. It's a one-time step. I pump in some currents. And all I want to know is know what are the potentials there. If I use a direct – a sparse direct method, and I send an interrupt signal to it, and I just say, "Nope. Sorry, sorry, sorry. Give me the voltages." It's like, "What do you – I – it has – it's not even – have you had –." It says, "I haven't even done, like, the – I haven't finished factorizing it, so I can't give you anything."

So – but CG, if you run CG on this, it's just doing iterations, and it's just like, "Well I was scheduled – I was gonna do a million iterations. I've only done three hundred, but sure, okay. No problem. Here's V." So these are – some people call these, like, anytime algorithms. That's one name for this, which is kind of dumb. It means you can interrupt at anytime and ask – I mean well, the contract is that if you interrupt it ahead of – before it declares that it's finished, you can hardly complain about what it returns. But it returns, like, an approximation. And, in fact, we'll see that for CG – and you'll even see why, in this case because you can analyze it. You're actually gonna get a shockingly good – and in fact, there's whole fields that are based on this where – which are actually quite funny. If you talk to people who do image restoration or they work in astronomy or something like that, they're huge and so you say, "What do you use?" "CG." And you go, "How do you actually do that?" And they go, "Yeah. We run, like, ten steps." And then you go, "Oh. Oh, that's you're method?" And they go, "Yeah." And I've actually had a conversation with a person who said, "Yeah. We run ten steps of CG." You know, it's like for geophysics or something like that. So they do ten steps of CG. And I said, "Well, why not 20?" And they go, "Oh. Because each step takes like, you know, an hour." These are 3-D, big, giant, huge things, right? Okay. So, each step takes – and I say, "Well, what happens if you run it 20 steps?" And they say, "Oh, then it starts looking bad." So basically, it's already in ten steps. At ten steps – this is for a problem with, like, ten to the nine variables, or something like that, right? In ten steps, you've already gotten some vague outline of what you're looking for. But the round off is already killing you after about 20 steps, or something like that. And I'm like, "And you guys are cool with that?" And they're like, "Oh, yeah. No problem. That's fine." They were like – so that's there prescription, which is run CG until you run out of time, or the round off is built up and the results start looking worse. I mean perfectly good method. Oh, I should say, here's the joke. I'll tell you something, so you will actually, in your homework, when we finally – when we figure it out and assign it. You'll run some CG things. And, in fact, there's a metlab PCG thing, which you'll use. And you'll be very thankful for it, even though it's a five-line method. And it, in fact, when it returns, if you say run CG 50 steps, what it returns by default is actually not the iteration after 50 steps, but in fact, the iteration between 1 and 50 that had the best residual.

So – and this is not like – this is not for the reason that you see in sub-gradient method, where it's not a descent method, where ahead of time you know, this is not a descent method. This is supposed to be a descent method. It's only not a descent method because of round off things and things behaving poorly. Okay. All right. So let's look at how this works. Well, the solution, we'll call it X star, that's A inverse B. And because A is positive definite, does it – when you – actually, there's a very useful thing. When you're solving AX equals B, where A is positive definite, a very good way to think of it is this. You are actually minimizing a convex quadratic function. And that fits, actually, in lots of applications. For example, if you're solving X equals B, and someone says, "Why are you solving that?" You say, "I'm going Newton's method." Someone says, "Explain to me Newton's method." Then you really realize, in fact, what you're really doing is you're minimizing this because Newton's method goes like this. You – someone says, "What's Newton's method?" And you say, "Well, I'm minimizing this weird function. I'm – I have a current estimate of X, and I go to the function and I get from a convex quadratic approximation. And I'm minimizing that." So in fact, this connection is not abstract. It comes up in the applications. Okay. Now, the gradient of this function is X minus B, but this is also – that's a residual. So a residual in gradient of this function over here is gonna be – they're kind of the same. Okay. Now, if you take F star is F of X star. That's this – I guess if you're – that's this thing plugged into that and – I forget what it is, but there's like an A inverse there, or something like that. You get an X – you get a B – it's minus B transposing inverse – anyway, it doesn't matter. But you take F of X minus F star, here's what you get. That is – here's F of X, here, and then minus X star. And you can rewrite that this way. And so it's actually X minus X star – this is the square of – and this sub symbol here means it's the A – people call that the A norm. So that's the A norm, and it means A positive definite. But this is sort of the distance in that metric. So F minus F star is half the squared A norm of the error. Yeah.

Student:[Inaudible].

Instructor (Stephen Boyd):What's that?

Student: The B transpose X stars?

Instructor (Stephen Boyd):Yeah? You mean –

Student: It just –

Instructor (Stephen Boyd):Here?

Student: No.

Instructor (Stephen Boyd):Here?

Student: Can it be a catalytic FX minus F star?

Instructor (Stephen Boyd):Yeah.

Student: Since X and X star aren't likely to be different.

Instructor (Stephen Boyd):Yeah.

Student:B transpose X terms –

Instructor (Stephen Boyd):You mean these don't cancel away?

Student: Yeah.

Instructor (Stephen Boyd):Is that what you're saying?

Student: Yes.

Instructor (Stephen Boyd):No. I don't think – I mean unless you're really good, I don't thing you can go from this line to this one by just looking at it. There's a whole bunch of lines out there commented out. No, no. I mean – yeah, look. But X star is this thing. So X star itself has a bunch of Bs in it. So when I throw – this term here, when it all expands out, is gonna have a giant pile of Bs as well.

Student: Can you transpose X star X transpose AX, or like –

Instructor (Stephen Boyd):Hey, that's a good point.

Student: Yeah.

Instructor (Stephen Boyd):So let me respond to your question by saying this is not – these are not meant to be self-evident. There are many commented lines out. And in fact, they may or may not be correct. They're – I'm going for like 80 – 80 – I'd go 85 percent on the reliability, maybe 90, you know? But I think they're right.

Student: Are you assuming A is positive definite then –

Instructor (Stephen Boyd):I am.

Student:[Inaudible].

Instructor (Stephen Boyd):In fact, A is positive definite until I tell you otherwise. Okay? So yeah, I mean otherwise, a lot of this doesn't make any sense at all. Okay. Okay. And now, a relative measure would be something like this, F of X minus X star divided by F of zero minus F star. And that would be something like that. That's tau. And it's actually the fraction of maximum possible reduction F compared to X equals zero. And actually, these things are very interesting. Here would be one. In a Newton method, what – this would have a beautiful interpretation. It makes perfect sense. So in a Newton method, it would go like this. You have a convex function. You're at a point. You form a quadratic approximation of that function, and then that goes down. And in fact, if you were to find the minimizer of that and take that step, that's the Newton step. And in fact, that height difference is lambda squared over two. That's the Newton decrement, if you remember that. So that tells you, sort of, if the quadratic approximation is right, it tells – it predicts how much you're gonna go down. If tau is a half, it says that you don't have the Newton direction, but you have a direction good enough that you will achieve one half of the reduction you would have gotten had you gotten the exact Newton step. So if you can actually calculate tau – which, sadly, you cannot – as you run these methods.

But if you could, it would be very useful because it would – because you – there's no reason you – you don't really care about the Newton direction, in fact. What you really care about is the direction that's gonna get you a good enough decrease, right? And something like if tau is a half, you could jump to the – you could quit and use that method – use that direction. Okay. So just say a little bit about the residual. By the way, I mean some of these are not too obscure. I mean so some of these multi-line things; one line should follow from the next, but not all of them. I mean without some paper and stuff like that. Okay. So this is the residual. It's the negative gradient. And you can write it this way. But in terms of this residual, you have F minus F star. It turns out to be one-half of the residual norm, but in the A inverse norm. Okay? So in fact, in all these methods you're gonna see, a lot of things are gonna be very naturally stated in terms of the A or A inverse norm. Now, sadly, this is generally not the norm you're interested in the residual, or something. It can be, in some applications. Generally, you're just interested in the actual residual norm, for example. And that's one of the things that's not – that you're not gonna have easy access to. But a very commonly used measure of the relative accuracy is the residual norm divided by B. So that's very common. By the way, B is the residual norm, if you guess X equals zero. Right? Because then AX minus B is minus B, or whatever. And so basically, if this number is bigger than one, it means that's a real – that's not a really good estimate of the solution of AX equals B because in fact, you're out-performed by a function, which is six characters that returns the vector zero, which is not – that's not a good place to be. That – anyway, so that's not an impressive performance. So this is usually how this – and it scales it with the problem size and things like that. Now, you can show that tau is less than the condition number of A times eta squared. So – and we'll see that – well, I mean here, eta is easy to – you know, I mean as the algorithm runs, you have X. And you'll see part of the algorithm is you're gonna calculate the residual at each step. You're gonna calculate the X minus B. Fine. You can take the norm of that, that's easy. You know B; you can take the norm of B. So tau, you're gonna get – sorry, eta you're gonna get very easily, and you'd get something like that. If A were well conditioned, this might be useful. So this is really – and this is maybe not so useful in practice, but it basically says that tau, which is arguably what you're really interested in, if eta is small, then tau is gonna be small too. Something like that. Of course, you don't know the condition number of A generally, so this is not that useful. Okay. Now, we get to the basic idea. So the basic idea is one of a – if has to do with Krumlauf subspace. There's – I – actually, who – anyone – who knows how to pronounce it Russian? So – because that's not how you pronounce it Russian, just to let you know. But it's been – I think it's been around for a long time. It, you know, we've been using since 1952. So now, it's Krumlauf subspace. It no longer refers to this guy's name. Okay, so – and that means that we have license to mispronounce it. So okay, so this comes up in a lot of places. A Krumlauf subspace is this.

It's the span of B, AB, up to AK minus one B. If you've taken a class on linear systems or something like that, it's also called the controllability subspace. And it's got other names in other fields, and I forget what they are. But it comes up in a bunch of other fields. That's called a Krumlauf subspace. And here's the cool part about it. It can be written this way. It is the set of vectors that can be represented as a polynomial of A times your vector B. Okay, where that polynomial has a degree less than K. So that – these are – it's the same thing. So you are – you're basically – you take polynomial of A, any polynomial of A up to certain degree, and multiply it by B. Okay? That's very important. It's – well, it's totally obvious because the linear combination of these is the linear combination of powers of A, then multiply it by B. Okay. So the Krumlauf sequence, X1, X2, and so on, is defined this way. It's the minimum of this F of X – that's that quadratic function – over the Kth Krumlauf subspace. So that's what this – that's what the – that's called the Krumlauf sequence for given A and B, right? So for example, the X1 is something like this. It's a long – the only direction you search on is B. And so you find – you adjust the co-efficient in front of B so you get alpha B minimizes norm AX minus B – no, the quadratic form, okay? So that's what you – you'd minimize that along there. And you get some multiple. So now, there's many methods to compute the Krumlauf sequence. Many. But the CG algorithm is the most famous one. But there's others, so – and it's actually worth not focusing on CG and actually focusing on the meaning of the Krumlauf subspace. So, okay. So that's the idea. All right. And now lets look at some properties. Well, your function goes down. That's – because you're minimizing a convex quadratic function over a bigger, bigger subspace. If I minimize a function over a bigger set, let alone subspace, the function value – the optimal function value can only go down. It can't go up. I mean it can stay the same, I guess. Okay. Now, if you go here all the way to N, you get X star. Okay? So and that's actually the case, even when the Krumlauf subspace does not fill out all of our N. So it is a fact. That comes from Cayley-Hamilton theorem that the solution of X, AX equals B is in this span of KN, the Krumlauf subspace. And let's give an example. Someone give me an example of a – when would a Krumlauf subspace have shockingly – not be – when you keep running them, you get – the dimension doesn't grow? What would be a good example? Oh here's one. How about this one? How about – well, here's an extreme would be B equals zero. So what are the Krumlauf subspaces when B equals zero?

Student:[Inaudible].

Instructor (Stephen Boyd):They're all just a point – they're single tints. They have their only element is zero. Okay, that goes to Krumlauf's – they are subspaces, right? Every thing's cool. And if you want to minimize that you just – this sequence is just zero, zero, zero, zero. This is when B equals zero. But here's the good news. The first one is a hit because the solution of AX equals zero is X equals zero. So it's okay. A more extreme example would be something like this. What if B were an eigen vector of A? Then what does the Krumlauf subspace look like? That's – it's just the first thing. You get B, and then you get a multiple. So the – actually, all the Ks are the same after K1. So K1 is K2. But again, no problem because then it say that after the first point in the Krumlauf's sequence actually solves it because you get – because it turns out, in that case, if AX equals B, and B is an eigen vector, then in fact X equals A inverse B, which is actually along the direction of B. But we'll get to that. So – but that's obvious anyway. Okay, so – okay. All right, now, this – the K Krumlauf sequence is a particular polynomial. That's a polynomial of degree less than K times B. Now, so far, every thing's actually fairly obvious. And now we get to the cool part, and it's not obvious. It's not obvious. And here it is. It turns out that to generate XK plus one, you can generate it from the previous two, like this. And then these are alpha K. This – in fact, it's just a linear occursion. These are numbers that are gonna come up. Their particular values are gonna come up. This is not obvious at all. And this relies on A being positive definite. That's it. So in general, if I just said that's the Krumlauf – here's the Krumlauf subspace, here. And I asked you to calculate XK plus one, you would actually have to look back at all – I mean just in general, an in fact, if A is not symmetric, you will actually have to look back over all of them. Okay? What that means is the effort to produce the next point in the Krumlauf subspace is actually not going to grow with K.

Student: So are you just taking the residual and getting the best approximation and the last?

Instructor (Stephen Boyd):No, you're not.

Student: You're not?

Instructor (Stephen Boyd):You will see what it is.

Student: Okay.

Instructor (Stephen Boyd):Yeah. It's not quite that. You'll see. So it's – these things are not totally obvious, and we'll get to what they are. So that's it. This – so if you wanna know, like, why is this interesting and all that sort of stuff, it all comes down this one statement. That's where the non-triviality is. Okay? So, okay. And just – oh, I think I mentioned this already, but the Caley-Hamilton theorem says that if you form the characteristic polynomial, like this, of the matrix, then it says you can write – if you – I mean then it says that that's zero. And therefore, will take A to B non-singular, A positive definite. So if you take all these terms – not that last one – and you pull an A out, for example, on the left. And then you put this on the other side, and you get this. You get an explicit formula. And this, I think if you took 263, we saw this. And I probably mentioned it. In fact, my forward reference was to this moment because I said there are – you know, most of the things I would say in a class like that, like we'll get to that later, we never do. I don't know if you noticed that. But in this case, we actually did, if you stuck it out this long. So, okay. So this says that the inverse of a matrix actually is a linear combination of powers of the matrix up to N. Okay? And in particular, it says that A inverse B is a polynomial of degree less than N of A times B. So it's in the Nth subspace, of whatever like that. Okay? So that's the idea. So even if the Krumlauf subspace, by the way, doesn't fill out all of our N, which, of course, in general it does. If you pick a random B and a random A or whatever, it's gonna fill out everything. Okay. So that's it. Okay. So now we're gonna look at the spectral analysis of the Krumlauf sequence. So what we'll do is this. We'll take A and we will write its eigen expansion, or spectral decomposition. And lambda's gonna be this diagonal matrix with the eigenvalues. And we'll define, you know, Y is Q transposed X. That's Y expanded in the Q basis. B bar is gonna be the same thing. Y star is gonna be that.

And basically, this means that the whole problem has now reduced to – everything's diagonal. Now, you know, this is not a method. This is a method for understanding it, just so you know. So it says this. F of X is F bar of Y bar, and I just plug in this thing. That's gonna be my – this is gonna be my Y and that's Y transpose. And so you end up with this. But now this thing is completely separable, and it looks like that. Okay? So. Yeah. I mean this is a bit silly, right? It says that if you want to solve AX equals B, and someone were kind enough to provide the eigen system, you could solve AX equals B very well because it would – you'd transform it. It would convert to a diagonal system, and that – solving AX equals B, ray is diagonal, that's easy. That's no problem. And so you'd be asked to minimize this and that, I presume, you can do. And so on. You'd just get – actually, this is – that's the formula. That's basically capital lambda inverse B bar. But capital lambda is diagonal. So you get this. Okay? So, so far, nothing interesting here, but now let's look at the Krumlauf sequence. Well, what you're doing here is this is your – YK, is the argument of this thing over this subspace here. But this subspace here is actually really simple. It starts with a vector B bar, and you keep multiplying by these diagonal matrices.

So these are just different diagonal matrices, and what that says is that the Ith component of Y at the Kth Krumlauf point is a polynomial, PK, times the eigenvalues, times BI bar. Okay? So that's it. And therefore, we can actually say we have now a beautiful formula, not in practice but for understanding. If you understand this, then you understand – you will understand all of it, of how it works. It says this. It says that the Kth polynomial – that's the polynomial that generates the Kth iterate of the Krumlauf sequence, when applied to A and then multiplied by B. That's the argument over all polynomials of degree less than K of this thing. Okay? That's a positive number. It's a weight. And now you look at this and you want to know what polynomials can – what polynomials can you do here? Oh, this thing, of course, will be – you know, this will be – this thing here can be negative. That's the – that's actually the whole point, because of this thing. And so here, it's lambda I times P of lambda I, and then minus P of lambda I. So in fact, well, we'll – this is P of lambda squared.

We'll get to what this means in just a second here. Okay? So you can write this another way. This – it says that if you look at how close you are to the Kth iterate to the maximum reduction you're gonna get, that's F bar of YK, minus F star. And that's the minimum of this. And now, these are just writing – reading it in different forms, so you start figuring out how you wanna – how to understand what happens. But don't worry, we're gonna go over this on next Tuesday, which, as I mentioned earlier, will probably be tomorrow. Next Tuesday. So this – if you simply work out what this is, you take out this thing here. And you end up with this. This is a generic polynomial here, whose degree is N here because – or degree is less – sorry, whose degree – this degree is less or equal to K, but it has it the property that if you plug in, let's see, zero in that polynomial, you're gonna get one. That's gonna be this thing. That's the negative of this. So I can rewrite this this way. And you finally get to something like this. And it's quite beautiful. It goes like this. It says that if you wanna know how close you are to the solution after K steps, you are exactly – there's no inequalities here – you're exactly this. It says, take BI bar squared, that's actually how – that's the – that's basically sort of like – in fact, BI divided by lambda I is the solution, or something like that. But it says, what you want to do is you want to find – you look at all polynomials of degree K, and then this sort of a weighted positive sum of Q evaluated at those points. Okay, now, by the way, you're not supposed to be following all the details. It's just sort of the basic idea here. And then, so what it comes down to is this. It says if there is a polynomial of degree K – ammonic – I'm sorry, not ammonic, but one that adds zero as one, so constant co-efficient one, that's small on the eigenvalues of A, then it means you've got a good approximation. Okay? And this tells – now, this tells you everything. So it works like this. It says, for example, if the eigenvalues of A are sort of all over the place, but let's just see some pictures about how this works. And then we'll quit for today. Let's – here's zero. And let's draw some eigenvalues. So if the eigenvalues look like that, and you have to have a polynomial that starts at one. And if I asked you sort of what's the best degree one polynomial – it means, best, means it should be small near these points. It's gonna be something that looks like this, right? Okay? But if I ask what's the best degree two polynomial, you're gonna make something that kind of – I could – let me just try. It's gonna be a quadratic.

And it's gonna kind of look like that. Everybody got that. And look at this. On here, you're sort of – the polynomial is not too – it's pretty small. So what that says, this thing then predicts that the second – if that' were your second spectrum, the second step in the Krumlauf sequence is gonna actually have really – I mean really low error, right? Everybody see this? This is very, and by the way, if it were exactly clustered, in other words if A had exactly, like, two eigenvalues, X2 will be the solution. Okay? That's not that interesting, but it's a fact. Everybody see why? Because then I can find a degree two polynomial that will go exactly between – that will just go right through and give you zero. And you get the answer. Does this make sense? Okay. So this is – I mean this just the first – your first exposure to it. We're gonna quit here. And in the next lecture, which like I said, will probably be taped ahead sometime tomorrow, we'll go on and look at all the implications.

[End of Audio]

Duration: 77 minutes

ConvexOptimizationII-Lecture13

Instructor (Stephen Boyd):Great, I guess this means we've started. So today, we'll continue with the conjugate gradient stuff. So last time – let me just review sort of where we were. It was actually yesterday, but logic, I mean, logically – in fact. But we can pretend it's five days or whatever it would be.

So we're looking at solving symmetric positive definite systems of equations and this would come up in Newton's method, it comes up in, you know, interior point methods, least squares, all these sorts of things. And last time we talked about, I mean, the CG Method the basic idea is it's a method which solves Ax=b where A is positive definite. And – but it does so in a different way.

The ways you've seen before are factor solve methods. In fact, in those methods what you need is you actually need the matrix. So you actually – you pass a pointed to an array of numbers, roughly. And then you work on that.

What's extremely interested about the CG method is actually the way A is described is completely different. You do not give a set of numbers. In fact, in most of the interesting applications of CG you will never form or store the matrix A, ever, because, in fact, in most cases it's huge. It's some vast thing. That's kind of the point. Instead, what you need is simply a method for calculating A times a vector.

So what you really are gonna pass into a CG method is a pointer to a function that evaluates this thing. How it evaluates it is it's business, it's none of your business. That's sort of how – that's the idea.

Okay, well, last time we started looking at a little bit about this. We looked at two different measures of the error. So one measure is this number tao and it's the amount of decrease, F is that quadratic function you're minimizing, you've achieved divided by – sorry, this is the amount of the decrease, the sub optimality and decrease divided by the original sub optimality of decrease.

That's probably what you're interested in. But another one which is probably in many applications what's actually easier to get a handle on and all that sort of stuff is the residual. And the residual is nothing but, you know, b-Ax. It's basically how far are you from solving Ax-b.

Now the CG method, and actually many others, actually work with the idea of a Krylov subspace. And this is just to sort of rapidly review what we did last time.

The Krylov sequence of a set of vectors is defined this way. It's actually – you take this Krylov subspace and that's the stand of b/ab up to some Ak-1b. And that essentially means it's all vectors that can be written this way. It's B times a polynomial of A and that polynomials of degree K minus one. That's a subspace of dimension, oh, it could be K but it actually can be less. Actually, if it's less than K then it means it you've solved the problem.

Okay, so XK is the minimum of this – this is the function you're minimizing, this quadratic function on the Krylov subspace, it's the minimizer of that. And the CG algorithm and several others generate the Krylov sequence. That's actually the important part.

Now, in the Krylov – along the Krylov sequence obviously this quadratic function that you're minimizing decreases. That's obvious because, in fact, you're minimizing over a bigger and bigger subspace in each step and that can't get worse. Now the residual can actually increase, that's not monotone.

Now it turns out if you run N steps you get X star, that follows from Cayley-Hamilton theorem. And the Kth iterate of the Krylov sequence is actually a polynomial of degree K-1 or less multiplied by B. Now the interesting part and the reason why these methods actually work really well – although the – by the way, there's plenty of cases where you're gonna do this for such a small number of steps that this is actually not that relevant.

There's a two term recurrence and the two term recurrence is this, you can compute the next point in the Krylov sequence actually as a linear combination of not just the previous one but the previous one and the one before that and then there's some coefficient here and we'll see what the coefficients are later. Actually, they're not that relevant. What matters is that they exist and are easily calculated.

Okay, so we've seen this and what I want to do now is look at the – to understand how the CG method works or how well it does. It's extremely important to get a good intuitive feel for how it works.

When is it gonna work well? By the way, notice that you would never even say anything like that when you talk about, like, you know, direct methods. You wouldn't say it's good to talk about, you know, let's say, "Well, here's a 1000 by 1000 matrix. Oh, yeah, here's one that's gonna work well." It always works well. You have positive definite matrix, you take a Cholesky factorization, it doesn't depend on the data. I mean, to first and – to 0 and 1st order does not depend on the data.

So okay. With these though you need to know when is it gonna work well because that's actually the key to all of these things. So the way to do this is essentially to diagonalize the system. Then when you diagonalize it's kind of stupid because if A is diagonal and I ask you, "How do you solve Ax=b?" that's easy. That's just the inverse, A is diagonal.

So the solution if I diagonalize is actually just this, it's just the transform B divided by the Ith entry in the transform A which is diagonal. This lambda thing here. So that's very simple. But this is gonna give us an idea for when this works well. The optimal value is just this. It’s nothing but this thing. That’s that A inverse B star, A inverse B. That’s this thing here. Okay?

Okay, now the Krylov sequence in terms of Y is the same except now we can actually look very carefully at this thing because the polynomial of a matrix is really, really simple. The matrix is diagonal so a polynomial of it is simply the polynomial it’s a diagonal matrix and each entry is a polynomial of that entry in the diagonal which are the eiganvalues of A.

So you get very simple expressions, all in terms of the eiganvalues now. So it says that PK is the – one way to say it is that it’s the polynomial of degree less than K that minimizes this sum.

And notice it’s got some positive weights, none negative weights here. And then over here you can kind of see what’s going on. If you look carefully at this you really want – well, we’ll say what P should look like in a minute. P should look like, you know, one over lambda or something like that to make this work well.

Okay, so another way to write it, let’s just keep going down here is we’ll look at the second expression. The second expression says that this error is the minimum over – these are the positive weights, and then here you can see it’s lambda I times lambda IP of lambda I minus one. And in fact what that says if you can make P of lambda look like one over lambda on the eigenvalues of A you’re – in fact if you could have P of lambda I equals one over lambda I it’s done. This is zero and then that says that in fact this would have to equal F star. Okay?

So that’s the idea. And in fact, these – we saw already in the Cayley-Hamilton theorem that there’s an Nth degree polynomial which in fact we saw exactly what PN is. It has to do with a characteristic polynomial. It gives – it’s a polynomial that in all cases satisfies P. The P of lambda I equals one over lambda I and that’s why this conversion ten steps – sorry, in N steps.

Now what’s interesting here is this is gonna give us sort of the intuition about when this works well. There are lots of other ways to say it. I mean, one is to say – well, look. A polynomial – something that looks like that is a general polynomial of degree K with the value that if you plug in it’s probably if you plug in 0 this goes away, you get one.

I mean, I switched the sign on it. So that’s another way to say it is this way. There are lots of these. I won’t go into too many of the details but the important part are the conclusions. So here are the conclusions. If you can find a polynomial of degree K that starts at one that’s small on the spectrum of A then the Kth – then actually no matter what the right-hand side B is you’re F of XK minus F star is gonna be small.

And in particular this says that if the eigenvalues are clustered into groups, let’s say K groups, then YK is gonna be a really good approximate solution because if I had K clusters of eigenvalues I can take a Kth order polynomial and put it right through, let’s say, the center of those clusters or near the center of those clusters and then that polynomial would be really small on each of those clusters and we'll get a very good fit here.

Now there's another way to do well and that's to have YI star small for just a handful of things. So this says if your solution is approximate linear combination of K eigenvectors then YK is a good approximate solution. So that's another way to say it.

Notice that this statement is independent of the right-hand side and depends only on the matrix A. This one now depends on the right-hand side but doesn't depend on the clustering. It basically says if you're a linear combination of K eigenvectors then this must be a good solution.

So, okay. Now you can do things like this, this allows you – these are classical bounds. Classical bounds would be things like this; you would – suppose the only thing I told you about A was that its condition number is kappa? So I give you – let's say I can scale A. So I give you lambda min and lambda max and if I put a Chebyshev polynomial on there, that's a polynomial that's small uniformly across it on that interval, you end up with a conclusion that says this, it says this convergence measure that this thing goes down like that.

And this is actually – this allows you to – oh, by the way, a simple gradient method would have a kappa here and a kappa here. So if you just use the gradient method to minimize that function F you'd get kappa here and kappa here. And so you're supposed to say, "Wow, that's much better because you get square root here," or something. So that's the idea.

It turns out – this is interesting and that's fine but it turns out actually that where you want to use CG is where in fact, I mean, this is like many other things it's an upper bound and in fact, you usually get convergence. It's sort of much, much faster. Not to high accuracy but –

So let's do an example here. So these show you – this is the function Q. They all start at 1 here. It's a 7 by 7 matrix. Now I think it goes without saying that CG is not the method of choice for solving a 7 by 7 positive definite system. That's something – I guess the time to actually solve a 7 by 7 positive system is down in the – it's definitely sub microsecond, you know, obviously. So this is not the right way.

Nevertheless, this is sort of the – we'll just look at an example. So here are the eigenvalues of A, I don't know, it's one here, I guess it's around two, a couple down here, another cluster here and an outlier out there.

After one step of CG the optimal Q is this thing. By the way, it goes without saying that you don't ever produce these. I mean, if you want keep track of these polynomials that's fine, but you don't need to.

So this shows you that one and you can see it. It's kind of doing its best. It's not so small here nor is it small here and it's pretty big there. The quadratic is this green one and you can see it kind of – it gets small near this cluster and it kind of splits this cluster and this cluster and goes near it so that you get things like this.

The cube term, I think that might be the purple one here, is the cubic term and you can see now cubic is nice because it's sort of – you can see three clusters and it does just what you think. It goes down, goes right through this thing. It's very small here, small here, small here, over here it's very, you know, quite small, still small, still small, and then kind of goes down there and it's small here. So actually you could expect that after three steps of this method you're gonna get a very good solution.

The quartic I think is the red, maybe, I guess maybe that's the red. I'm not sure. I guess that's the quartic or something. And the quartic one as you can see goes through – is now actually picking off bits and pieces. It's actually doing things like hitting both sides of it so it's small on all three here. It's small here and now it's just nailing that one.

And then the seventh one is this one and that's actually an interpolating polynomial so it's 0 on all of them and that means that at step seven you get the exact answer. Okay? So, I mean, actually I'm anthropomorphizing this, obviously. So well, all that's actually done is the Krylov sequence is computed. But this gives you a rough idea of how this works.

Okay, actually you'll know shortly why it is that you need to know how well CG works because it's gonna be your job to change coordinates to make CG work. We'll see that in a minute.

Okay, so here's the convergence and sure enough, you know, you start with that's the full decrease, and you can see this sort of after five steps you've done very well and I guess after four you've done extremely well and so on. So that's the picture. Okay, here's the residual which in fact does go down monotonically. It didn't have to but it did in this case. Now look at – I mean, that's a fake example. Let's look at a real one.

Here's an example, it's just a resistor network with 100,000 nodes. We just made – it's a random graph so each node is connected to an average of ten other nodes. So, you know, some big complicated resistor network.

So, again, a million nonzeros in the matrix G. And I pick one node and I make that the ground. Okay? Then I'm gonna do the following, I'm gonna inject at each of the million nodes a uniform current, a current that's chosen randomly uniform on 01 and the resistor values will be uniform on 01. Okay?

So that's our – that's the problem. It doesn't matter. But in this case if you want to use a sparse Cholesky factorization – actually, before you ever do it you'll know what happens because you can actually do the symbolic factorization on G and actually calculate the number of fill-ins. Okay?

So in this case it required – there was too much fill-in and I don't know – I don't know how big it would be but maybe, I don't know, 50 million, 100 million or something like that, nonzeros starting from 1 million. Okay?

So – oh, I should mention this, if the number of nonzeros goes up by a factor of a 2 you are to consider yourself lucky in a sparse matrix thing, right? And that means you must go and make an offering to the god that controls the heuristic of sparse orderings and of ordering in sparse matrix methods.

Ten, you know, that's okay. You're supposed to be happy or that's typical. You start getting to fill-in factors like 100 and stuff like that and that's because you didn't go make an offering the last time sparse matrices went your way.

So, all right. So in this case this problem you can't solve with a sparse Cholesky factorization and I actually shouldn't say that. I should say using the approximate minimum degree ordering method produces a Cholesky factor with too much fill-in.

Now, of course, on the big machine I actually could have done it and would have gotten the exact answer but it would have been really long and taken a lot of time. It might be that there's another heuristic ordering method that would work perfectly well here. I doubt it but anyway, there's lots of them.

Okay, so instead we'll use CG here. Now in this case I do form the matrix G explicitly and all I have to do at each step is I have to multiply by G. But that's just a sparse matrix vector multiply is a million nonzeros so I'm doing like a million flops per matrix vector multiply and that's a dominant effort of a CG iteration. So I don't know, how much time does that take?

Student: About a second.

Instructor (Stephen Boyd):A second. Man, we've got to work on you guys. This is not cool.

Student:[Inaudible].

Instructor (Stephen Boyd):What?

Student: Less than a millisecond?

Instructor (Stephen Boyd):What?

Student: Less than a millisecond.

Instructor (Stephen Boyd):Thank you, less than a millisecond. So around a – let's just say a millisecond and let's just get the order of magnitude right, millisecond. Okay? So matrix vector multiplies a millisecond here, roughly. Is that right? Thousandths of – yeah, sounds about right, right? Yeah, sure. Weird. Isn't that strange? Man, these computers are fast. Okay, that shocks me.

All right, so it's a millisecond, matrix vector multiply. And it might be a few milliseconds because of all sorts of, you know, issues with accessing memory and stuff like that. But if it's set up right and you're lucky it's on that order of magnitude.

Okay, so here's how CG runs and this is a residual here. So – and you can see that, well – for ten steps the residual actually goes up by a factor by 100 – generally considered not good. And then it goes back down again. But the wild thing is – the theory tell us the following, the theory says that if you run it one – what did I - was 100,000? Yeah.

The theory says if you run it 100,000, you know, the millisecond doesn't sound right to me but I'll have to think about that. I think that's – I have to do that for each one or something? Anyway, maybe it is right. It doesn't matter.

The theory tells you that K here runs out to 10 to the 5; we'll get the solution exactly. But the wild part is, is if you're not picky about super high accuracy you actually have a perfectly good solution in 50 steps.

Each step was a matrix vector multiply and if our estimate of a millisecond is right it means you just solved a very large, you know, diffusion equation, Poisson equation, whatever you want to call it. You just solved it in a quarter second. I mean, if we're right or, you know, something like a tenth of a second. Everybody see what's going on here? By the way, absolutely nothing in theory guaranteed that this would happen, absolutely nothing. Okay? It just did. And this is very common.

So – and this is kind of the cool part about CG is that in a shockingly small number of steps you often – what emerges is something that looks kind of like the solution.

In this case it doesn't look like the solution; it's actually, like, quite good. Okay? So that's the idea. So if you wanted to know at what point have you learned something new, you just did. You cannot – if you go just type this thing, you know, G\I or whatever, and let a sparse, you know, even if you have a big computer it's gonna take a long time and a lot of RAM. And then this'll just get a pretty good answer in 50 steps and just be way, way shorter. Okay?

Okay, so here is the CG algorithm. There are many variations on it. People seem to focus more on the algorithm itself than actually on the – what the algorithm produces. In my opinion it's much more important what the algorithm produces which is the Krylov sequence and as many – there are other methods to produce the Krylov sequence and they have different round up properties and things like that.

Let me show you what those are. So here's one and this is – instead of just sort of making up my own I just followed exactly one from a very good book on the subject by Kelly. So just to make it – not to invent new notation or anything, it's this.

And the only important part you need to know here is something like this; you maintain the square of the residual at each step. If the residual is small enough, you quit. So this quitting criterion epsilon is on the ratio, it's what we call ada; it's on the ratio of the current residual to the norm of B. That's this thing.

And what you do now is something like this. Your search direction is gonna be something like P and it's a combination of both the current residual and the previous search direction. Then all of the effort in here, well, not necessarily but in most cases is right there, this one thing right here. This is where you call the mult by A method is called right here.

Everything else if you look here is actually sort of an O of N or a blass level one or however you want to call it, call. So, for example, here you update a vector that's O of N, that's O of N, you have to calculate an inner product.

Actually, if you want to parallelize this that's the one that is really irritating. Everything else here can be done in a – is completely distributed. So the main effort here is this call to A. These other things I guess this is also has to be collected together so that's another one that would not be distributed easily but that's the algorithm.

By the way, these calculations can be rearranged like 50 different ways and so you get different versions of it. In exact arithmetic all of those ways will compute exactly the same sequence XK. With round off errors in there they can be different and you'll find people talking about one versus the other and this that and the other thing and you'll find all sorts of different flavors and things like that and people telling you one way is better than another and all that. Yeah.

Student: Is there anyway to know, like, if it will be faster than a theoretic convergence?

Instructor (Stephen Boyd):No, I think the theory just gives you, like, rough guidelines and basically says if you're – if the eigenvalues are clustered – for example, if they're tight, if the condition number of the matrix is small that condition number [inaudible] will tell you it's gonna nail it in 50 steps. Okay?

If they're spread out but have, you know, clusters or something like that, it's gonna nail it, that kind of thing. So in general you don't know. Kind of the worst things you can have are ones whose spectrum is sort of uniformly spread all over the place, right? That's the kind of the worst thing.

Now it also depends on the right-hand side. So if the right-hand side – the B, actually if that one – the worst thing that could happen is B can be sort of a uniform mixture of all of the eigenvectors and that would be kind of the worst thing to happen or something like that.

Okay, so let's talk about – so as I said at the beginning this is mostly interesting. I mean there's a fundamental difference between this and a direct method. In a direct method you give the matrix A, you give the coefficients to the algorithm literally. You pass an array or some data structure and you get the entries.

In CG you do not need that. All you need is a method that multiplies by your matrix A. There's nothing else you need. Okay? That actually in interesting cases that can be some, like, specialized hardware, it could actually carry out an actual experiment. I mean, it could do – it could solve the whole PDE, it could do insane things, right? Do a full up simulation of something, run Monte Carlo, it could be all sorts of weird stuff.

But it doesn't have to be a matrix, is the point. So here's some examples, why can't you do an efficient matrix spectrum multiply. This is kind of obvious, if A is sparse, for example, if it's sparse plus low rank now you better know the factorization here.

And if you want some numbers here you should think of A as like a million by a million. That's just kind of, you know, because if A is, I don't know, 10,000 or something this is kind of not worth it or something. Yeah, so you should – your mental image is should be that A's a million by million or 10 million by 10 million. That's a good number. A 100 million by 100 million.

These are the numbers you want to think about when you think of CG and how these things work. If you have products of easy to multiply matrices, that works. Fast transforms come up so FFT, Wavelet, DCT, these types of things, things like fast Gauss transform that actually is when A looks like this and you do this via multiple pull methods.

This is actually an extremely interesting one. Here's a matrix that is extremely – that is dense, well, half dense, and that this the inverse of a lower triangular matrix, in fact, even the inverse of a sparse lower triangular matrix, that's a great example.

So I give you a sparse lower triangular matrix, by the way, the inverse of that is generally completely dense. So what – and I couldn't even store – let's make it a million by million, in fact, let's make it a million by million banded.

The Cholesky factorization of a million by million banded, of course, I could solve that directly, but anyway. A million by million banded or something like that is actually gonna be banded. It's gonna have – it's small, it's easy to do. The inverse is gonna be a full lower triangular matrix. You can't even store it and it would be completely foolish to actually calculate the inverse matrix and then multiply each time.

But if I give you a vector and I ask you to multiply by the inverse, it means you have to solve a lower triangular set of equations and that you can do by back substitution. If it's a – I'm assuming – I'm taking the case where it's sparse.

Okay, so these are just examples. It's very good to think of examples like this. Okay, now couple things you can do. You can also shift things around. So if you have a guess X hat of the solution X star then, in fact, what you can do is actually – well, you can – this is the residual with your guess. If you solve Az equals this residual then your solution is actually gonna be just if you solve this you get the optimal Z and you add it to X hat and that's your solution.

What happens now is that XK now looks like this. It's actually X hat plus this and it's the argment of this quadratic over the shifted Krylov subspace so you shift by this X hat. And there's nothing you have to do to make this happen, all you need to do is initialize not with 0 in the first step but with X hat and everything will work.

Now this is also very important because this is good for worm start. So what this means is, if you need to solve, let's say a giant circuit equation, giant resister equation, if that's part of integrating a circuit, I mean, sorry, integrating a differential equation for a circuit you will step forward one couple of picoseconds or whatever and you'll solve a similar system. You can use the last thing you just calculated as your guess and this can often give you, like, stunningly good results. So as you will see an application of that in optimization soon.

Okay, but now we come to the real way it's used and I should say this. I should say that if you – in most cases if you simply run CG on a problem it won't work.

The theory says it will work but that's not relevant for several reasons. It's not relevant number one because you have no intention of doing a million – if you have a million dimensional system you have no intention of running a million steps because you don't have that much time. That's the first thing.

So to say that it works in practice means that it works in some very modest number of steps. Or a lot of people I remember hearing – I hear this in – here's something you hear on the street that you're doing on CG when you're doing – pretty well when you're doing square root N steps. Theory says you have to do N but the word on the streets is square root N should do the trick for you.

And if you think carefully about what that means it's just a rule-of-thumb it says if you have a million variables in a thousand steps you should have a pretty good solution. Okay? These are just – I'm just saying these – it could be slow, you know, but you shouldn't be shocked if it's on the order of squared N. That should be the target. So 100 million, 10,000, that kind of thing. Okay? These are the numbers.

Okay, and the other reason it's not that relevant is the following – the theory is that errors in – when you're doing conjugant gradients round off errors they actually add, it's unstable and the thing diverges, actually, very quickly.

So by itself CG generally – often will just fail. If you just have some problem and you run it, it's just gonna fail. Okay, so the trick in CG is actually changing coordinates and then running CG on the changed – the system in the changed coordinates. That's the trick.

That’s called preconditioning and now you know why it is that you needed to know when CG works because the key idea now is to change coordinates to make the spectrum of A, for example, to make the spectrum of A clustered or something like that or live in a small interval. So that’s the key.

So here’s the basic idea, you’re gonna change coordinates, you’re gonna use Y is the coordinates of X and some T expansion and what’s gonna happen is you’re gonna solve this equation here and then in the end set X stars to inverse Y.

Then sometimes people call T the preconditioner but then sometimes they call TT transpose the preconditioner and you get all combinations of preconditioner meaning T, T transpose, M, T inverse and T minus transpose. So there may be some other possibilities like M inverse but the point is that anytime anybody says preconditioner you have to look very carefully and figure out exactly what they mean.

Okay, so that’s the preconditioner. And in a – basically what happens is when you – if you rerun – if you just take CG it’s actually not that bad. You have to multiply by T and T transpose at each step. The other option is to multiply simply by M once and you don’t really need this final solve, in fact.

So this is called PCG. And by the way, this is exactly where you – this is why in these iterative methods you have room for personal expression, right? So if you’re doing – if you’re solving dense equations there is no room for personal expression. If you do anything other than get glass and run Atlas to [inaudible] or cash sizes and things like that and then write good code then it’s just not right.

There is no reason under any circumstances to do anything else. You go to sparse matrices – actually, in sparse matrices is there room for personal expression if you’re doing sparse matrix methods? Yeah, and what is it?

Student: Ordering.

Instructor (Stephen Boyd):Ordering, yeah. So there is some room for personal expression in sparse matrices and as to selection of ordering and that’s cool. Actually you can say, “Oh, I know my problem. I know a better ordering than approximate minimum degree is finding” or something like that. Or there’s exotic ordering methods. You can go look at Netus is a whole giant repository at Minnesota that’s got all sorts of partitioning methods and some of those are rumored to work really well.

Okay, but we move up the scale one step higher to iterative methods. Now, boy, is there room for personal expression and it is mostly expressed through the preconditioner. So, in fact, when you get into these things you’ll find everything is about finding a good preconditioner for your problem. And then you can go nuts and you can have simple preconditioners and complex ones, unbelievably complex ones and things like that. So that’s the – there’s a lot to do.

Okay, so the choice of preconditioner is this. Here's what you want. You want the following. You want a matrix T for which T transposed AT or a matrix M, let's say M. I want a matrix M for which the eigenvalues of MA are clustered, for example.

And here's one choice, how about this? The inverse. That'll do it because now the eigenvalues of MA are all 1; CG takes one step because the eigenvalues are all 1. It's kind of stupid though because you actually then have to, like, sort of invert the matrix or something like that.

So that doesn't make any sense. So here's what you really want. What you really want is a matrix M which somehow approximately is some kind of approximate – well, see approximate can be very crude. An approximate inverse of the matrix and yet is fast to multiply. That's what you want. Okay? So that's the essence of it. And so this is the idea. And the M – this M can be quite – you'll see approximate it like very generous here. I mean, you can be way, way off.

Here's some examples. The most famous one and often very effective is diagonal preconditioning. It's kind of stupid but actually you should try that always and immediately first because it often just works. So that's – you would not call the diagonal of the inverse of the matrix or whatever you would not call that an excellent approximation of the inverse. But you have to admit it's cheap to multiply and all that kind of stuff and it works quite well, amazing.

Here's another huge family, it's actually really cool and it works – it's called incomplete or approximate Cholesky factorization. And so here's how that works.

Now what I'm gonna do is I'm gonna compute a Cholesky factorization not of A but of some A hat. And the way you might do it is something – it's weird. It might work like this, you might run a Cholesky factorization, in fact, there's a whole field on this called incomplete – it's also called ILU, incomplete LU factorization. These are preconditions based on this.

And the way it would work is this, you take the matrix and you start doing a Cholesky factorization on it but you might decide if an entry is too small when you do the Cholesky factorization – you say, "Oh, screw it. Just forget it. I'm just gonna ignore it." Or if you're doing something and you're gonna fill in in various places which is the death of direct methods, you just say, "Forget it. I'll just ignore it."

So you end up calculating a sparse – well, it's a sparse Cholesky factor or something but it's definitely not the original matrix, right? So these methods can also work really, really well, these things.

And here will be an example; you can do the central KY band. That's a version of this. It's a version of this, too, where you just basically say, any fill in of L below some band I refuse to even – I won't even go there. So let's see, these are some obvious – anyway these are sort of the obvious things.

These can work really well and a good example would be something like this, if you have a problem where there's a natural – where something is ordered, for example, in space or in time like in signal processing or control you have time. Then what happens is, you know, things are connected locally, you know, states, transitions, signals and things like that and that leads to this banded system.

Now banded systems you can solve super fast, we all know that. But supposing you have a dynamic system or signal processing but a few things were kind of – you had a main bandwidth of, like, 10 or 20 and then a light sprinkling of little things all over the place everywhere else.

So for some weird reason in your problem, you know, the state here is related to the – it's just, I don't know, I'm just making this up. But the point is you could easily make up an example like that. Everybody see what I'm saying? So it's kind of banded plus a little light sprinkling of nonzero entries other places, okay?

So this would work unbelievably well. You simply do a banded – you simply ignore all the crap off some band, you do a Cholesky factorization on that, that's your preconditioner, and the only thing you're working the error to fix is all that little light sprinkling of entries that were outside that band. So that's the idea.

By the way, a lot of this stuff that I'm talking about, I mean, these are whole fields, I mean, this is the basis of all scientific computing, PDE's, all – so there's tons of people that know all of this stuff backwards and forwards. A lot of this stuff though hasn't diffused to other groups for some reason. So these are actually just really good things to know about.

Okay, so here's that same example with 100,000 nodes and a million resistor circuit and here it's just with diagonal preconditioning. I mean, it's kind of silly because it was already working unbelievably well but you can see this is what diagonal preconditioning does. Diagonal preconditioning actually is mostly useful not for this kind of speed up but for when this residual goes like this and goes like that and then it would go like this, okay.

Theory says it doesn't do that, that's all from round off error. So that how the – and in fact, here's a very common CG stopping criterion used on the street, this. You run CG until the answer starts getting worse then you stop, then you just say, "Well, that's it, sorry. Here's what it is."

So – and I know that's done and I know people who do that in image processing and computational astronomy and all sorts of things. They just – they run CG until it starts – that usually only gets it a couple hundred steps and then basically you can keep running CG but things are getting worse because of the numerical errors you've lost or foganailty and all sorts of things happen.

Okay, so here's the summary of CG. Actually, the summary's all that matters. So here it is, in theory, and that means with exact arithmetic, it's not particularly relevant. It converges to a solution and N steps, period.

That's not interesting or relevant because N you should think of is on the order of a million roughly and the whole point is you have no intension of doing a million steps. Your goal is to do it in a thousand steps, couple hundred, whatever. So that's kind of your goal. I mean, if you're really lucky, 10, 20, 30, these are the numbers you really want.

Okay, now the bad news is that if you – is that actual numeric round off errors actually makes this thing work much worse than you would predict. Now the good news though is that with luck that means a good spectrum of A you can get good approximate solution in much less than N steps. Right?

So now the other main difference with a CG type method or something like that is this, and this is very important, you never form or need the matrix A. Anyway, you can't store a million by million dense matrix anyway.

Well, I mean, maybe you could but the point is you don't want to. So the point is you don't need the matrix, you don't need the coefficients the way it works for direct methods where you pass in a data structure and it's got all the coefficients in it. Okay?

Here you don't need it. What you need only a method to do matrix multiplication. That means you have to rethink in your head all – everything you knew about linear algebra and you have to rethink to this model where what you really have is a matrix vector multiply oracle and that's it.

What the oracle does, how it's implemented is none of your business. You can call it, you can give a vector and it will come back with AZ. And by the way, you could – this would be very bad, but you could actually get the coefficients in A from an oracle. How would you do that actually?

Student: Multiplying by –

Instructor (Stephen Boyd):Yeah, EI.

Student:– EI.

Instructor (Stephen Boyd):Exactly. So you call the oracle with E1 and what comes back as the first column of A? Then you do it for E2, second column, and actually after EN you've got all of A. So then you could pack it in there and then call it LA pack, whatever. But that's not the point here.

Okay, and it's very important to fix in your mind how this is different from factor solve methods, okay? It's less reliable, it's data dependent. So, for example, that circuit I showed you that was with a random topology. I might take another circuit that's got like a pinch point or something in it, 10,000 iterations, nothing happens or it's even worse, so it diverged.

These are data dependent, okay? And this is not the case roughly speaking for direct methods. They're data dependent. And – but there is – you can either think this is the bad way or the good way, the bad way is this, these methods don't work in general unless you change coordinates with a clever preconditioner. That's the bad way.

The good way is to say is that CG methods have lots – I mean, the employment prospects are very positive because you don't just call a CG method, you need somebody to sit there, figure out the problem, try a bunch of preconditioners and tune things and find out what works. So that's – you can think of that a good way. There's room for considerable personal expression in CG methods.

On the other hand you can hardly complain, this is really your only method for solving a problem with a million, 10 million, 100 million variables. So if you made the mistake of going into field where that's needed then that's your problem.

Student: What – so if you have any operator form it's a very, very hard to get some of the preconditioners you mentioned before. You just use easier ones?

Instructor (Stephen Boyd):Yes, so that's – okay, yeah. So if A is an operator form usually you know the operate – I mean, typically you know the operator so it's not this thing where you don't even know what the oracle does. The theory says you could do that.

You know, to get – well, no. So for that you have to sneak around the oracle. It's a good point, you know, you have your – it's not a pure oracle protocol. Just to get to diagonal you'd sort of sneak – you'd send a messenger around in back of the protocol and say, "Would you mind telling me what your diagonal is?"

Normally a reasonable oracle will be willing to tell you the diagonal. So – but that's a very good point. If you have to find your diagonal by calls to the oracle you're screwed. Yeah, okay, that's – any other questions about this? Because we'll go on to the next – the topic which is actually just a – it's kind of obvious it's just applying this to optimization.

Okay, so the – these are called truncated Newton Methods. Actually, there's a lot of other names for it, we'll get to those in a bit. And they're called approximated, truncated and you can do this all the way up to interior point methods. So, let's see how this works. So here's Newton's method and in Newton's method you want to minimize a smooth convex function, second derivatives.

And so what you do is you form – you want to calculate the Newton step by solving this Newton system here. It's symmetric positive definite, that's symmetric positive definite. And you might do that using a Cholesky factorization. This would be the standard method, you know, for problems that are either dense with up to a couple thousands variables or problems that go up to 10,000 or even 100,000 something like that but where the sparcity is such that the Cholesky factorization can be carried out.

So that's what you do here. And then you do a backtracking line search on the function value. It could be on the function value or the norm, either way, and you'd stop basically based on the Newton decrement, that's this thing, or the norm of the gradient. So that would be your typical stopping criterion.

Okay, so an approximate or inexact Newton Method goes like this, instead of solving that Newton system exactly we're gonna solve it approximately. So – and the argument there is something like this, you don't really have to get the Newton direction exactly, that's kind of silly. In fact, for sort of convergence you only need a dissent direction.

Of course, the whole point – the advantage of the Newton Method is these – especially for these smooth convex functions is that it works so unbelievably well and you get your final quadratic conversions and things like that.

So what you really want is to trade off two things. What you want to do is you want to do a fast and sort of crappy job of approximately computing the Newton steps. You want to get a fast delta X. But if it's too crappy a job then the algorithm is actually gonna make slow progress.

By the way, as to whether or not it's gonna make progress at all it'll always make progress because you will see that almost any method for approximately solving a Newton step, every single step is gonna generate a direction which is a dissent direction.

And so convergence of the methods, at least theoretically is guaranteed. What you can lose is you'll lose things like quadratic terminal convergence and things like that. Now if you're solving gigantic problems you may not be interested in quadratic terminal convergence, you're interested in just solving it even to modest accuracy but in, you know, the amount of time, for example, less than a human lifetime. I mean, that's your – but again, this is your problem for solving such big problems. This is your fault, I should say.

Okay, so this is sort of the idea. Now here are some examples, one is this, is to simply replace the Hessian with a diagonal or another one is a band of this and use that as the Newton step because if that's diagonal or banded this can be solved incredibly efficiently. So that's one method.

Another method, and I think we even explored these in 364a a bit or maybe you did a homework problem on this or something. I can't remember. If you didn't, you should have. So – you did? Okay.

Another one is to factor, this is called the Shimansky Method is you factor the Hessian, every K iterations and then use that for a while. And, of course, that requires a method – a factor solve method.

Now the factor solve method – oh, that just reminds me, I have to say something. In a factor solve method if it's dense or in some dense factor solve methods there's a big difference between the factor and the solve. The factor typically goes like N cubed and the solve goes like N squared. Right?

So what that says is you get weird things you can say. You can say things like this, "I need to solve Ax=b." And you say, "No, actually I need to solve Ax=b, Ay=, you know, Ax2=b2, Ax3." You want to solve it multiple times with multiple right-hand sides.

In a dense factor solve method it's the same cost because the expensive part was the factorization. Once you've factorized the – you can solve – you put an investment in and you can now solve multiple versions of that problem very cheaply. That's the key behind this. Okay?

Iterative methods, nothing like that, none. Iterative – want to hear the cost of solving Ax=b by an iterative method if that's like C? Then the cost of solving Ax=b and Ax~=b~ along with the other one? 2C. There's no speed up. You simply call the solver again. Everybody see what I'm saying? So some things that you probably just got used to thinking about, for example, if you factor back solves are essentially free because they're in order less, now don't transfer to these iterative things.

Okay, so this is the Shimansky Method. It also works quite well. Okay, truncated Newton Method is very, very simple. It does this, you're gonna run either CG or PCG and you're gonna terminate early and in fact, it's very important to understand the key is not to terminate early, the key actually is to terminate way early. That's kind of the hope here.

That's – if you want to get stunning results it's gonna have to be something like that. Now, these also have lots of other names. They're called Newton Iterative Methods, they're called – it's also called Limited Memory Newton. It's also called Limited Memory BFGS. You end up with exactly the same – you have to go through all this horrible equations, everyone has their different notation system, but in the end you find out it's sort of the same method. So these methods have lots of names.

Okay, now in a situation like that the cost is not the cost – the cost per iteration is completely irrelevant. So the cost is measured by the number of CG – actually, it's the number of calls you make to the multiplied by the Hessian. That's actually – that's the exact cost is that. That's the number of CG steps.

And to make these things work well you're gonna need to tune the CG stopping criterion, you want to use just enough steps to get a good enough search direction. If you use too many CG steps then you're gonna get a nicer search direction but it's gonna take you longer to get there.

If you use way too many CG steps you're gonna basically be doing Newton's Method now. That's great because now, you know, whatever it is, 12 steps, it's all over. But each of those steps is gonna involve 4000 CG iterations or something. At the other extreme if you have too few steps you’re basically doing gradient, it’s gonna jam and it’s gonna take a long time.

Okay, now it’s, of course, less reliable than Newton’s Method which is essentially completely reliable. But, you know, again, with good tuning, good preconditioner, a fast Hessian multiply and you need some luck on your problem, you can handle very large problem.

I should say that what – I should say something else about these methods. Whereas one can write a general purpose convex optimization solver and you’ve been using multiple ones, you’ve been using SDPT3, [inaudible], all these things.

If you think about it that’s really very impressive. I mean basically they don’t know what problem is gonna be thrown at them. They can be scaled horribly. I mean, you can make it too horrible it’s not gonna work but they can be scaled horribly, all sorts of weird things. You can have weird sick, you know, flat feasible sets and all sorts of weird stuff people throw at these things every day and they do a pretty good job. They’re general purpose. Okay?

So but when you get into these huge systems it’s much more ad-hoc and ad-hoc means literally it means everything is built, you know, for this.

So, although people have tried to come up with sort of general purpose iterative methods and they’re getting close with some things, I think, in some cases. But generally speaking what this – you’re gonna need to tune stuff. You’re gonna need to – I mean, it has to be for a particular problem.

You have to say, “I’m interested in total variation denoising.” Or, “I’m interested in this pet estimation problem.” You know, in medical imaging or something. I mean, it has to be a specific problem. “I’m interested in this flow – aircraft flow scheduling problem.” Okay?

So having – now the good news is this, once you’ve fixed to a particular – it’s, by the way, not a particular instance of the problem but the problem class. Once you’ve fixed to one of those things I am not aware of any case where these methods cannot be made to work.

Obviously that’s not a general statement. I can’t back that up by anything but every time I’ve ever looked at any problem, as long as you say it’s a specific thing, like it’s a network flow problem, it’s a this problem, it’s a gate sizing problem, it’s a problem in finance or machine learning, these work always. They don’t work in 15 minutes, they work in – it takes a while, right? Maybe not a while, so that’s kind of the good news of these methods.

So – and it’s kind of the answer to this, maybe once a day somebody comes up to me and whines, often by email actually, because they’re not here. But they whine, they go, “I have my problem CVX, you know, I can't solve it" and blah, blah, blah. And okay, you're like, "It worked fine for 10,000 variables, it doesn't work for 100,000." And I'm like, "Well, first of all it's mat lab. It's made so you could rapidly prototype your problem. You need to solve 100,000 variables, you need to know how these things work and stuff like this."

So this is the answer to people like that. They're like, "No, I don't want to write my own software, it's too hard" and everything like that. I'm like, "Then don't solve problems with 100,000 variables." Just go away, don't bother me, or something.

So – but the bottom line is you want to solve a problem with 10 million variables, 100 million in a specific area, like a specific little area, it's gonna work basically. So it's gonna require some luck and it's not gonna happen in 15 minutes.

Okay, so here's truncated Newton Method. You'll do a backtracking line search on this, you can do a backtracking line search on F as well. The typical CG termination rule is gonna stop when this is your Newton System residual here, divided by the gradient, which by the way is the right-hand side. So this is exactly the ada that we had before. This is an okay one, it says – all of these things are kind of heuristic and stuff like that. So exactly what you use to stop maybe doesn't matter so much.

And what you'll do is you'll have simple – with simple rules you'll iter out at some constant number and you'll have this epsilon PCG, that's constant. And so this might be like .1. That would be a reasonable number there. You wouldn't want to put .01 and you sure as hell would not want to put 1E-4, 1E-4 says you're basically calculating the Newton direction and there's no need for that.

Well, maybe to get the thing up and running and verify your code works and stuff like that, you might start with a small problem and make this 1E-4 just to make sure you're actually calculating the Newton step and it should look then exactly like a Newton trace at that point. But you might want this to be .1.

So a more sophisticated method would adapt the maximum number or this thing is the algorithm proceeds and the argument would go something like this. It would say that, look, early on you don't need to solve the – the whole point of Newton is that it changes coordinates correctly in the end game so that if you're stuck in some sort of little canyon instead of bouncing across canyon walls as you would in the gradient method you're skewed towards a direct shot at the minimum. That is what Newton's Method is.

So you'd argue that actually Newton's Method at first you'd totally – I mean, in many cases you're wasting your time. So then you might have actually at first this might be, like, very – this might start .1 and then it might kind of get – go smaller or something and that might modulate depending on how close you are to the solution. There's a lot of lore on this and you can find this in books and things like that but a lot of it is just to play with it. So that's it.

You would find some theory on this and it's, you know, it's mildly interesting and things like that. I mean, this would – you'd find some – go find some thing from the '70s or '80s or something like that or in some book that would tell you, "If you did this you'd guarantee super linear convergence." I guess my response to that is, you know, we're not really trying to achieve super linear convergence; we're trying to solve problems with 10 million variables.

Student: Right. So you [inaudible] tends to take, like, a long time. Is there –

Instructor (Stephen Boyd):It shouldn't. Why would it take a long time?

Student: Because essentially, I guess, it just starts, like, has to do a lot of queries to get to the, like, I guess –

Instructor (Stephen Boyd):It shouldn't.

Student:– it [inaudible] with your criteria for –

Instructor (Stephen Boyd):It should not.

Student: It should?

Instructor (Stephen Boyd):Yeah, a general rule of thumb is, you know, you said beta equals a half, so your step – every time you query you're divided by equals two. If you do five or six of those that's too many. And the average number of lines or steps you should be doing is like three or something, two, no more, often one point something.

So I'm suspicious of that. And I'm just telling you sort of what people experience. Yeah, that's not – and in any case you have to evaluate F. You're gonna evaluate it at every step here. Not every CG step, right? But every outer iteration you're gonna evaluate the gradient of F and I haven't added that in to the cost here.

So, yeah, that would come up. But, yeah, you shouldn't be doing a lot of line search thing. Right? I mean, first of all, Newton's best when you get into the end game you're doing none. I mean, if it's really Newton you should be doing none.

And that will translate over here. When you get down in the region of quadratic convergence with a method like this you shouldn't be – even though you don't have the exact with the Newton direction which is aiming you right to the solution, it's a good enough direction that a full step should be taken.

So it's actually kind of a bad sign if you're doing more than a couple of Newton steps. And if later in the algorithm – I'm sorry, backtracking a second. Later in the algorithm if you're doing lots of backtracking steps it doesn't sound right. Is that a specific problem you're thinking of?

Student: It's just – yeah, I guess.

Instructor (Stephen Boyd):Okay, we'll talk about it later. Okay, now you also have the question of CG initialization. One way is to initialize with delta X=0. It turns out in that case the first step if you go back and look at the CG thing is you – well, actually it's very simple, you solve Ax=b. In the first step you minimize over the line span through B. B in this case is minus the gradient.

So the first – after one step of CG to solve a Newton Method if you start from 0 what pops out is a very interesting thing. It's something – it's along the negative gradient direction.

So if you take CG and you said N max=1 which means do only one step then it turns out that the steps popped out by CG in fact are scaled versions of the negative gradient. So – and in fact, you can prove things like this, this is – that might even be a homework problem on that in which case we'll assign it to you or maybe – I forgot. Or maybe there're – if there's not a word problem maybe there should be one or something like that.

You can prove the following, that when you run Newton's Method, sorry, CG to do an approximate Newton Method every step of CG takes your angle of your search direction closer to the Newton direction.

Everybody see what I'm saying? So let me show you the – how that works. So here is – let's just make it quadratic. I mean, it doesn't really matter. Here you are, you don't know it's quadratic, let's say. And you are right here, okay? Now the negative gradient direction says search in that direction, right?

But the Newton direction says, "No, I think actually that's a better step." That's – well, that's Newton, right? If it's quadratic it nails it in one step. And what you can show is this, that if you run CG starting from 0 your first step will be in this direction and every other step is gonna actually close – is simply – they're gonna simply move over here in angle towards the Newton thing. Okay?

So everybody see the idea here? So basically you bend – you can even think of CG as sort of, you know, modifying your step taking into account more and more of the curvature, in fact. That's a good model for what it is.

Okay, now another option is this, you can choose – you can actually use the previous one and do a warm start. Now one problem with that is that if you start with zero every step of CG it will be a dissent direction and this might not be the case here. But you can give it an advantage if you're only gonna do ten steps, if that's what your tuning suggestion, you can actually do really well by just using the previous one.

And the simple scheme is something like this, if the previous step is a dissent direction for the current point use it, initialize this way, otherwise you initialize from zero and these are just sort of schemes.

So let's look at an example. It is L2-regularized logistic regression. So here's my problem. It's gonna look like this, and if you want to know what this is, is I am fitting the coefficients in a logistic model right here and there's L1-regularization over here.

So – and what that means is you have a logistic model, you have binary Boolean outcome and I have a vector of features and I have a giant pile of data that says, "Here are the features and here's the outcome like plus or minus 1." You know, like the patient died or didn't and I give you – or, you know, the stock price went up or it went down.

And I give you – let's say, oh, to make it interesting a million features. Okay? So – and I give you as a thousand samples. Now obviously that's horrendously over fit, you're over fit by a 1000 to 1. If I give you a 1000 samples a thousand patient records and a million features.

So what the L – hey, wait a minute here. Sorry, I'm gonna have to go back and cut out that whole discussion. I thought that was L1. So, all right. We'll just rewind and skip all that, sorry. We just gonna do L2-regularized. Actually, you can solve L1 – if you can do this one, you can do that one. So, all right. We'll just do L2-regularized. Sorry, pardon me.

L1-regularized would allow you to do it with a million features. It would run and it would come back with 37 features and it would say, "Here are the 37 out of the million that look interesting. These are the ones that I can make a logistic model and predict these thousand medical outcomes well" or something like that. Okay?

Sorry, this is not that. This is just an example. Okay, all right. So the Hessian and gradient looked like this. I mean, they start looking very familiar, it's [inaudible] equals DA plus 2 times lambda. Lambda is diagonal here. The gradient looks very similar.

And none of the details matter but the important point is all you have to be able to do is multiply by the Hessian, multiply the vector by the Hessian. When you do that you have to do this and the key is here you don't even form this matrix because this thing it's easy to make up cases where I can store A but I can't store A transposed DA let alone can I even think about factorizing it, that's a joke.

But this is a case where I can't even store it let alone factorize it. But anyway, I make this – I multiply it this way so I need two methods, I need a method – by the way, this often corresponds to a forward model method and that's a reverse or adjoint call. So I have a forward model call and an adjoint call and I'm gonna make two. For each I'm gonna make a forward model call and an adjoint call per CG step.

And we'll just make an example here with 10,000 features and 20,000 samples. So here we just may have a problem where these XI's have random sparcity pattern. They have around 10 nonzero entries and then in the nonzero entries it just shows at random. None of this matters. But you end up with about, I guess half a million nonzeros in this and we made this small enough that we could actually do the symbolic factorization just to know what we're talking here.

And so the factorization gave us 30 million nonzeros in the Cholesky factor. Oh, by the way, that's something we can handle but as you can see this method will be like just orders of magnitude different.

Okay, so the method is Newton which we'll look at in a minute and then truncated Newton with various things for stopping. Basically we're max iterating out because we're asking for .01 percent error and here's the convergence versus iteration.

So these are actually iterations and the Newton and the 250 step CG are right on top of each other. So the whole problem takes like – well, I mean, it depends what your accuracy is but, you know, it takes like 15 steps. This is what we expect from our friend Newton, right? This is exactly the kind of thing we expect.

Now, by the way, the cost of a Newton step versus the cost of 250 CG's is like just orders of magnitude off. The Cholesky factor had 30 million nonzeros. Okay? The CG – the original problem had something like 100,000 nonzeros or something like that. So you're just way, way, way off by orders of magnitude.

The time for these – you can see here that if you do only 50 CG steps you start losing a little bit, but not much. Now the ten is actually – this is very typical. What happens in the ten is you're making excellent progress and then down here that's this one, you actually – you stall. And the theory says that'll keep going down but we don't have time to wait for it. So that's the idea.

Now the right way to do this is actually to look at the convergence versus cumulative CG steps and I see a totally different thing. Oh, by the way, Newton step would be like, I don't know, probably over in my office over there, the first Newton step in terms of effort would probably be, I don't know, we'd have to check but probably way, way, way, a kilometer that way. Okay? Just to put these things in perspective.

Now, let's see here, what you can see if very, very cool. This guy that does ten CG steps, which is hardly gonna win at any prizes for, like, good approximation of the Newton step is doing, like, amazingly well and then it just jams.

By the way, if you're happy with a gradient ten to the minus five or whatever, like, this is ridiculous. Cumulative number of CG steps is 100 and so you're solving a problem that I don't know how long it would take with the direct method but this one might be in the order of like 20-30 minutes or something like that. You're now back to solving it in milliseconds. So or you're just orders and orders of magnitude faster.

So here – oh, here's some time so we can actually get the times. So, let's see, so if you do 50 or 250 you're basically the same. So here's Newton Method with N max and it jams near this gradient 10 to the minus 6 but that often is just fine.

Oh, so here are the time, it's 1600 as opposed – 1600 seconds so it's not bad, it's half an hour, something like that, as opposed to four seconds, okay? So these numbers are – these are pretty good factors here. These are worth going after these factors. So these are just much, much, much faster methods.

That's – yeah, this one that jams or something like that. So but you're already way off. I mean, you're down in the sub, sub, sub minute range. If you just do diagonal PCG here's the same picture and I think now it's simple. Now there's absolutely no doubt what the best thing to do is and it's right here. You just do diagonally preconditioned CG to do your Newton method.

Ten steps. Now that's ridiculous. If someone says, "What are you doing?" You say, "I'm doing a crappy job on Newton's Method." And you say, "Really? How's it crappy?" And you go, "I'm using an iterative method to calculate the search direction." You say, "Well, what's your iterative method?"

You say, "I'm doing ten steps of CG and then I quit. Then whatever I have at that point I pass it to the hieroalgram to do a line search." Then you say, "Oh, do you have some theory that says that worked?" And you go, "Oh, yeah, if I were to do 100,000 steps in exact arithmetic I would have computed the Newton step."

So, I mean no one can possibly justify why ten steps is enough here. I mean, you just can't do it, so that's it. But these things work done now unbelievably well, and now that means you can scale. That means you can do a problem with 100 million variables like no problem.

So here are the numbers for this and they're kind of silly. It takes 1600 and then it goes down to 3 seconds or it would be 5 seconds as opposed to, like, 45 minutes or whatever that is. So these are real factors here.

And by the way, if you make the problem bigger this goes up much faster than – this will just go up linearly so you'll just – if you were to make it ten times bigger you could solve a problem – and this would be a minute and that would probably scale much worse than linearly.

So okay, and you can use this for many, many, many, many things. You can use it for barrier and primal-dual methods and I think what we're gonna do is quit here and sort of – and finish up this lecture next time. So we'll quit here.

[End of Audio]

Duration: 75 minutes

ConvexOptimizationII-Lecture14

Instructor (Stephen Boyd):How's yours going?

Student: We've got lines.

Instructor (Stephen Boyd):What? You've got lines as far as [inaudible]? You did?

Student: No, as far as random.

Instructor (Stephen Boyd):Oh, as far as random, I meant. That's what I meant to say.

Student:[Inaudible].

Instructor (Stephen Boyd):Let me – no, let me find that. Are you guys gonna tell me when we're on because I've noticed the last couple of things – we're on now? All right. Well, good thing I didn't say anything I shouldn't have, not that that was happening. Sorry, we'll continue that discussion later. All right.

Well, a couple of announcements. Let's see. Oh, the first one is right after class today I'll have office hours. I feel guilty since I was in New Mexico last Tuesday. And actually, as it happens, I'll be in Zurich next Tuesday. So and, in fact, we're gonna tape ahead next Tuesday's lecture tomorrow.

I think it's like 12:50 to 2:05 here. So I feel like I might as well just bring in like a cot and stay in this room because I don't know. It would just be easier, right? A little espresso machine? Anyway, so tomorrow 12:50 to 2:05, that's on the website.

The other announcement I wanted to make was that the midterm progress reports or something for the projects are due tomorrow, so and do feel free with any of those to wave them under in front of me or either TA just so we can – because we can quickly look at it, scan it, and say, "Take it away," or point to things because we – I actually think really the more feedback you get, the better instead of just sort of – I don't know.

Basically, we don't want to read stuff we don't like. That's sort of – so I'd rather look at it for 30 seconds, catch a problem instantly, and just say, "Take it away and have it come back later." Anyway, so those are due tomorrow. Let's see. I can't think of any other sort of administrative details.

I guess you have a homework due today, and we're gonna make up another one and put it out like, I don't know, maybe today or tomorrow or something like that. There will probably be only one more homework after that one, so, and the idea is we'll just have like one exercise or something on each of the topics we're gonna cover just so that if we cover something you can't say you didn't do a problem on it.

So yes, you'll run a CG. You'll have to run a CG thing. We're trying to arrange these to be incredibly short so that you write ten lines of code simply so that you can say, "Yes, I've solved a problem with 100,000 variables in .2 seconds or something like that." Just so you can say you've done it.

Let's see. The other – oh, and another thing I was gonna say was that we're gonna settle on the last week. Some evening the last week we're gonna have a poster session for the projects. So the projects are due the Friday before the last week. I think that's May 30th. Is that true? My God, I think that's true actually because that's – isn't that like two weeks? Oh, my God. What happened?

Okay, so anyway, that's when the projects are due. You know, if there's any sort of emergencies, I mean we made it intentionally a week before the end of the quarter, so there's a little bit of slack there, not a lot because I'll be disappearing soon after the quarter finishes. But we will have a poster session the last week of classes. That's like June, roughly June 2. I don't remember exactly when it is, but we're gonna have a poster session.

So, and it's gonna be one evening. It'll either be like the Monday, Tuesday, or Wednesday, and we'll figure that out, and you'll hear more about that. Actually, if there's – if anyone knows of some reason why they can't – if there's some other event one of those evenings, someone should let us know quickly before we schedule it. But we're gonna schedule it. We'll get poster boards. Well, we'll get the posters from the department.

I asked the department if they would provide dinner for us. You can guess what the answer was, so maybe we'll figure out some way to cover dinner or something like that, or something like that for one of those evenings. So anyway, just plan that last week, the first couple of days in the last week we're gonna have a poster session, and it'll be just the standard 12. We'll put more stuff on the website, but it shouldn't be a surprise. I think all of you know what a poster is anyway. So it's the canonical form. It'll be 12 slides.

Your project should not be that long, so, and in fact, if they are organized, if the expedition is organized, it should take you like no time to go from a six page final project report to a poster. In fact, it should be – if it's organized, the sections are right, you state sort of what it's about, what the problem is, what are the attributes of the problem, it should just go perfectly. So anyway, you'll hear more about that.

Okay, so I think that's all I have for administrative stuff, and I guess we'll finish up this topic of truncated Newton methods, and then we'll go on to truncated Newton interior point methods. So we'll finish up another chunk of the class today, which is – and then we'll move on to yet another chunk. The chunk we're gonna finish up today is basically how to solve extremely large problems, so okay.

So let's look at – let's go back to truncated Newton methods. So remember what truncated Newton method is. It's you form a – you're minimizing a smooth, non-linear convex function, and you form the Newton system, but instead of solving for the search direction exactly by, for example, a direct method that would do some kind of direct factorization or something like that. Instead of doing that, what you'll do is use CG or PCG, preconditioned CG, to approximately solve the Newton system.

Now the way this is done is a bit surprising. What you don't do is actually spend all this – you don't do enough CG iterations to actually get a good approximate direct solution of the Newton system because there's no point because, in fact, your goal is not to solve the Newton system. Your goal is to minimize this function.

So, in fact, these things work really well, and, in fact, they work well precisely when a reasonably good search direction emerges after a shockingly few number of CG steps. So that's generally how these things work well. Okay, so this is an example. I think it's a logistic progression example with LS regularization. It really doesn't matter because it's kind of similar no matter what problem it is.

So this is what we'll do. The problem was scaled in such a way that we could pull off a Cholesky factorization. I think the Cholesky factor had something like 30 million nonzeros. So it'll take some time to do both the Cholesky factorization and also to do the backward and forward substitution. So but direct is possible. All we have to do with this problem is scale it by a factor of ten and direct becomes kind of out of the question, so then at least on a little standard machine.

But we'll compare that with the truncated Newton method where we do several things here. What we're gonna do is we're gonna max – we're gonna simply do ten steps. Now this says we'll exit if we actually do solve the Newton system within .01 percent. Is that right? Yeah, .01 percent, or if we hit ten iterations, so almost always this is gonna be ten iterations.

The second one essentially does 50 iterations, although we will terminate early if we solve the Newton system. And the third one will do 250 steps. Now I guess in the problem I forget how many variables there are, but I think it's tens of thousands or something like that. I actually don't remember. Well, here we go. Yeah, it's on the order of tens of thousands, okay. So maybe it is 10,000.

So it's 10,000 is the size of N, so the – let's bear in mind that the theory say that if you do 10,000 CG steps you will get the Newton direction. So by the way, the practice does not conform to the theory. The theory tells you to get the exact Newton step if you did 10,000 steps in perfect arithmetic, in exact arithmetic, which of course we don't. All these are done in doubles obviously. So but the point is this number is shockingly shy of the number that theory tells you will do the trick, and these are like a big joke, right? That's one-thousandth of the number of steps required.

Now on the other hand if you remember all this business about the spectrum, that's gonna be important. So the question is really the 10,000 says that no matter what the right-hand side is for any A, you're gonna get the exact solution. But the point is in ten steps you'll get a reasonable solution provided certain things happen. And if the spectrum is clustered, if the right-hand side, which is the gradient lies in the – a lot of it lies in the tenth [inaudible] subspace. That'll do the trick. Several things will work, okay.

So here's how it works. Here are the iterations, and let's see, here's the Newton method, which does what we all would expect it to do, which is basically nail this thing in 16 steps to some accuracy way beyond anything you need. So that's our friend Newton here. By the way, that's Newton applied to a smooth convex function, so Newton doesn't work this way in general, but okay.

If you do 50 steps, let's see. Oh, the CG 250 is right on top of it here. So if each of these – what that says is that the quality of search direction you get in 250 CG steps is whatever that search direction is, although no one would claim it is the Newton direction. Whatever it is, it is obviously good enough to provide – exactly match the performance of Newton method, right?

So this alone is gonna beat Newton's method insanely, both in memory and in time. But the really interesting part is if you do 50 CG steps you can see you're not – you know, you're down here, it's charging you one more, so that's even more shocking because that is now one-fifth the effort of this one. And the real shocker is this one where you do just ten steps. The ten step one actually, if you truncate around here, which is actually probably a perfectly good accuracy, is actually quite – is amazingly good. I mean this is actually sort of amazing up here.

Okay, now the real measure in a method like this is absolutely not the number of iterations. That's totally irrelevant. What's relevant is the number of CG steps because the CG step is what makes – it makes a call to a Hessian multiply method, right? And I mean exactly what I said.

You generally don't form the Hessian and then multiply it by a matrix. You're welcomed to do that. There's cases where you would do that, but remember that the interface of a truncated Newton method to the problem is simply you need something that will multiply the Hessian by a vector period. That's all you need. You don't need anything else.

So in this case, and this, by the way, would correspond extremely well with this time. So this tells you how fast it is. And you can see something shocking that ten steps of CG here gives you – if that's a good enough accuracy for you, it does this in just a shockingly small number of steps. So that's CG.

This is very common. This is – I guess it's called – sort of on the streets it's called jamming. The theory would tell you that if you did things in infinite precision, which you don't, that this would eventually converge. Of course, it might take a long, long, long time or something like that. We don't do things in infinite precision. We do things in doubles, and so in fact, it might just state here. In any case, for all practical matters, this is just – you're just wasting your time out here.

Yeah, by the way, this is gonna be a number so small that it – we're talking – well, we'll see in a minute what the ratio, how much faster this is than a direct method. So you see actually everything you see that is typical. This is quite typical to get extremely good conversions. The width of each of these guys is ten steps. The width of these is 50. They could be a little shorter than 50. They would be a little shorter than 50 if basically you solve the Newton direction.

Now what I should have done is I should have shown what happens if you do CG1. I don't know why I didn't put it on the spot, but CG1 is basically a scaled – it's not even a scaled gradient method. It's a gradient method. And CG1 actually goes like this, maybe goes like that and then stops like that, okay, so which is to say it's quite useless, okay.

So here – I won't go – we looked at this last time, but these will just give you some numbers here, but the ratios are pretty good. There are things like speed-ups of 400 to 1 here. And let's actually see what happens if you put PCG in. PCG is what you need – a preconditioner is what you need to sort of eliminate that jamming type thing. And so if you put in a preconditioner, what happens, it's not too surprising, is now your CG thing, despite the fact that you're doing ten CG steps, only ten CG steps.

You're actually getting a more than good enough search direction here to get extremely good performance in some just shocking number of steps. By the way, that's whatever is has, like 100 and something steps. Each step is basically a matrix vector multiply which you don't even implement as a matrix vector multiply in most cases. In fact, that's the whole point. So this is just a shocking number of – a shockingly small number of steps to do this.

Okay, so and if you want to see sort of what the numbers are there, they're not bad. They're things like 400 to 1 or something like that. These are good speed-ups. By the way, here the only thing we did was we didn't really want to wait more than an hour or something here, so that's the only – we set the problem size so that this would be like an hour.

The same problem, which I guess you have all the source code for because it's on the website. I promise you you could go in, make it ten times bigger, and the CG stuff will run just fine, so and then you'll be solving problems with a couple hundred thousand variables. You can almost certainly take it to a million and it'll just run. And it'll probably take – maybe it won't take 200 PCG passes, but it'll take something like 300 or something like that, something just amazingly modest, and it will work.

So okay, so there's a lot of extensions here. You can do the things with equality constraints. If you use an infeasible start Newton method but equality constraints, you have to – your equality constraints are actually not gonna be guaranteed to hold, right, because that's a property of – if you take a step of one in a Newton method, your equality constraints hold. In general, if you do a truncated Newton, it's gonna just simply converge to zero.

And, of course, you can take this idea and put it into a barrier method or your favorite or more sophisticated primal dual methods and so on. And I should also say that the distinction between direct and indirect methods is a nice gray boundary because in fact the ones you've been using, STPT3 and Sedume, when they – they use nominally direct methods. In effect, you can even see the right-hand column when you run CBX you'll see – we didn't run. I mean this is someone else's code, but in all of those things, they'll actually do several steps of iterative refinement.

What that means if they'll do a Cholesky factorization, but they will presume that there are numerical errors in it, and they'll simply use that as the preconditioner, and they'll do like two or three steps. So there's a sense in which you've actually already been using CG, but at the other extreme.

This is where you calculate an extremely good approximation of the solution and you have an amazingly good preconditioner. The things I've been showing you are at the other end where you are computing. You're doing a really bad job of solving a gross approximation of the Newton direction, but it's good enough to make progress overall.

Okay, so the last thing we're gonna do is just apply these ideas to interior point methods. So what's gonna happen, I mean you're gonna do – it's gonna be tricky to do this, but it can work well, and that requires luck, tuning, and a good preconditioner. So those are actually generally bad qualities for an algorithm to have, right. Imagine if every time you called or [inaudible] or A backslash B, there were 14 parameters you had to adjust, and you had to say, "Oh, A backslash B is not working." And someone says, "No, I know how to do it."

You have to go back and set A a little bit higher for problems like this, and then watch it, and it runs for a while and then jams, and then you have to come back and you go, "No, restart it and set alpha to .01 and see what happens." I mean so these are bad attributes. Actually, they're good attributes from an employment perspective because it means whoever needs to run these things is gonna need to pay someone to be there to baby-sit it.

So these large-scale methods, they will need tuning at a minimum. Once they're tuned, they may not need babysitting and stuff like that, but they're – so just to make it clear, these are not good attributes of methods. On the other hand, you know, you shouldn't complain if you're solving a problem with 10 million variables or 100 million variables. So that was – you made the mistake by going into a field where such big problems had to be solved. So you can hardly complain them that you have to tune and baby. And the generic software off the web doesn't just work for you.

So okay, so we'll look at network break control. This we've seen a whole bunch of times, and the problem is this. You want to maximize a logarithmic utility. So that's maximize the product of a bunch of flow rates. I have a matrix R, which has got 01 entries, and it tells you if you sum, RF calculates the sum on F. Each row of R actually gives you the links. It's got ones in the places where that flow goes over that link here.

So and then that's a capacity. So this thing is the total traffic. It's a vector of the traffic on each link, and that's the capacity on the link. And so the idea is a whole bunch of flows are sharing resources. They're sharing the links. The constraint is the capacity on the links, and we want to work out the sharing in such a way that this utility is maximized. This is the negative utility here. So that's the problem. It's sort of a classical. Actually, we've already seen this problem for distributed methods. We've seen that.

The dual rate control problem looks like this. We've also already seen this. You maximize N minus C transpose lambda plus, and then you get another log term here. And that's subject to lambda positive. Now remember the distributive method, so distributed rate control works like this. It solves this dual problem in a distributed way using a – well, using a projected subgradient method, although, of course, the objective is differentiable, so it's really a projected gradient method although that means something else, and it's not the algorithm we looked at before.

And the duality gap is simply given by this. If you have F – if you have both a primal feasible flow and if you have a dual feasible lambda, then this is the duality gap. So the primal dual search direction, if you just worked out what it is, it doesn't matter because all of these search directions end up looking the same. So if you use a primal barrier, dual barrier, all of these things, if you apply a Newton method, they all look the same. They might be a little bit different and everything, but they all have the same components.

Actually, they all look like this, a diagonal plus R transposed D2 R, just period, and that's for all methods. This is gonna be for primal dual method. What will happen is the D's might change a little bit, the right-hand side might change a little bit and so on, but otherwise they're gonna be the same.

All right. So once you solve this to get delta F. This is a block elimination method, and then you get delta lambda this way. And then these are just the details for this primal dual search direction. You don't have to. You can just as well use a – and you can just use a primal method. It makes no difference.

The primal dual residual is this. You break it into two parts. That's the dual residual and that's the centering residual, and then the algorithm just looks like this. This is the standard one. If you solve this exactly, it would be a primal dual method, and you'd expect it to converge in whatever, 30, the usual, 50 steps. But instead, we're gonna solve this system here approximately using PCG, and then you do a line search on the norm of the residual, which is standard, and so on.

Okay, so here's a problem instance with, let's see, 200,000 links and 100,000 flows. So that's a reasonable sized problem. And there's about – each flow has a length of about 12, so each flow goes over 12 hops, 12 links in the network, and each link has got about 6 flows on average going through it. We made the capacities random from over a range of 10 to 1, from .1 to 1.

Here the truncated Newton method is done with the following. You stop at either 10 percent accuracy, or that's the approximate duality gap divided by the dual function level, and you have an end – this actually, in this case, end max was never reached. So by the way, you might wonder where this is. This is kind of – when you enter into how to make these things work, how to set these things, there's a lot of lore, so it's a lot of room for personal expression.

So when you use these methods all the art comes in in figuring out a good stopping criterion for CG. And the idea is completely obvious. What you want to do is you want to do the smallest number of CG steps. You want to do the crappiest job of calculating a search direction that you can that'll make the overall master algorithm work reasonably well.

So and you always start by looking at smaller problems and doing lots of CG steps because if you can't make that work, then you're in trouble, big trouble. So once that's working, then you start backing off and start pushing things to the limit. And then only later when you dare do you get down to things like numbers of CG steps like ten and stuff like that.

Okay, this is a diagonal preconditioner. It uses warm start, meaning that it starts – it actually starts obviously from the last search direction, and then these are the parameters here. So here we're gonna go to 10 to the -4 [inaudible] which really isn't needed, but anyway. And here it is, so these are minute U of F. So these are the primal values and the dual values coming up like this.

And what's really ridiculous about this is in 100 PCG steps you basically have a very good – you basically have optimized the flows. You certainly optimized them within, you know, in a couple of hundred CG steps. Each CG step, by the way, is actually a forward. It's actually you multiply by R and R transposed. Those have physical meanings. You know, one sums for each flow, it sums some numbers across the flow, and then R transposed simply calculates the traffic on an edge. So it's literally 200 passes and it's all over. So you – 200 passes up and down and it's globally optimized.

So okay, this shows you the duality gap evolution here, and you can see actually that it's actually slowing down. Of course, if you do a primal dual method you expect to see – with exact Newton you expect to see something that looks like that. This is doing the opposite. It's slowing down.

By the way, I think this duality gap is already pretty small. The reason is this is a total duality gap, what you should really report is the duality gap per flow, and there's 100,000 flows roughly. So this is actually like 10 to the -8 here already or something like that. That's the duality gap per flow, meaning that I'm not even sure why we're showing this, but it was kind of all over about right here, so but okay.

Here's the same problem with a million edges and 2 million flows. So these are real problems. These are big problems now. These are not – you can't type this into CBX or something like that, and they're big, so and that's the same thing. It's just to show that it actually works and so on, and here's the duality gap, and again, I guess it's slowing down or something like that, but it's good enough. This is completely ridiculous, right, the idea that you've got an extremely good solution, provably so in 200 passes up and down across the network. I mean that's really quite ridiculous.

So okay, that finishes up this. I should say a little bit about this. I mean there's a lot of art on this, and if you look, if you just go to Google and type in any of these types of strings, other names would be limited memory, BFGS, and it's got all sorts of names, truncated Newton, iterative Newton.

If you type in any of these things, you'll find you'll start looking at some of the lower. And sometimes they flatly contradict each other. They'll be – one book will say you should make your stopping correction for PCG's go this way, and another one says it should grow. I mean so clearly no one has any idea, and you just deal with each problem on its own, so okay.

I should say one other thing is that I personally am no expert on large-scale optimization, but just for fun a couple of years ago we decided we would do some just to see how it works. And I can say the following. I am reasonably comfortable saying now I don't know of a problem where – we've never failed, ever.

So we did a couple of machine learning problems, some finance problems, some control problems, did some – oh, let's see what else did we do? Circuit sizing and stuff like that. Yeah, the details differed every time. Took one grad student week, that's a unit of time, a unit of effort, I should say, to tune things, but no problem. I would just come in one day and some grad student, when I would come in in the morning, they'd be leaving, and they'd say, "Good news. This PCG update rule nails it across a huge number of samples."

And so anyway, so I think the bottom line on this is pretty simple to say. It's just this. If you have a problem that's in the million variable, 10 million, of course, all of this, if you want to go much higher, no problem. But everything's gonna come down to your ability to do this matrix, to actually – well, to multiply it by the Hessian.

So you want to get a huge super gigantic problem, and you want to take 1,000 of your closest friend's processors, and make some giant NPI thing. I don't' see any reason why it won't work. I just don't see why it won't go to 100 million variables or something like that.

So then it's nice to know that if you have a problem and you solve small instances of it with CVX, and you like what you see, and it looks good and promising, it's nice to know there's actually a path to full large-scale things. So and I think that's really the takeaway message here.

Okay, this finishes up one whole section of the course. It is on solving giant problems on a single machine, although it doesn't have to be a single machine, in a sort of centralized way. It is actually centralized because the CG algorithm is centralized.

There's inner products are the parts that are centralized, and but I don't think it's that big a deal if – I mean if, I don't know, someone wanted to do this. It's not a big deal. These are just – they're scatter gather operations. And if you're only doing a couple hundred, I don't see why you couldn't do this for like really, really big problems.

So I do know someone who solved a billion variable LP just for fun, so not using these methods, used direct methods, but it took an entire – it took one of these blue jean, you know, it's an entire room full of computers. And each iteration, I think, took something – it was either like an hour or two hours per iteration. And I asked him how many, you know, what algorithm he used. Same one, primal dual, everybody else's. How many steps did it take? And he goes, "Same, 21." So but that's like a day and a half or whatever it was, so anyway.

So I think it's no reason to believe that these things won't go to much, much larger things. If you need to go to these large systems, I think it will work. Okay, any questions about this stuff? Like I said, we'll arrange that you will have person experience with CG, so and we'll arrange for your personal experience to involve nothing more than just a handful of lines. But you'll do CG, you'll run it, you'll solve something big, and then you can say you solved CG, or you solved the giant system using an iterative method.

I'm just curious actually. How many people have seen this stuff in other context? In where did you see it?

Student:[Inaudible].

Instructor (Stephen Boyd):Oh, sure, yeah. That's like whole courses, right?

Student: Yeah.

Instructor (Stephen Boyd):Okay.

Student: Finance.

Instructor (Stephen Boyd):In finance, really? Which class?

Student:[Inaudible].

Instructor (Stephen Boyd):Oh, okay, at KTH.

Student:[Inaudible].

Instructor (Stephen Boyd):Oh, [inaudible], sorry. I'm mixing up my Swedish students. Really? So CG for truncated Newton and just CG or just CG?

Student:DFGS.

Instructor (Stephen Boyd):DFGS, oh, okay, for some class on finance. That's cool. Okay, all right. The rest of you should just know about it. So maybe you'll never use it, but you'll do one, and then some day five years from now there'll be some giant problem and you'll be banging your head, and you'll look up and there will be a big light bulb and it will say in the middle CG. And you'll get to use it for something. You'll get to use it for something, sure, okay. All right.

Next, now we're gonna do – another chunk of the class is gonna be on a – I mean it's a family of methods that's been around for a long time. These things go back into the 50s, and someday when I'm bored, I'm gonna go back and actually find some of these things, these references. But it's like a lot of other things like interior point methods as well.

What happens is that for unknown reasons relating to marketing and just who knows what, fashions. All of the sudden, even though some exist for 30 years, the right person goes to the right place and somehow each person there tells ten of their friends, and then they each tell ten of their friends. And this happens three more times, and all of the sudden, this thing bursts on the scene as if it was brand new.

So that's the case for L1 norm methods for scarcity. You should know that it is in super duper fashion right now in many fields, in statistics, in machine learning, it goes on. And, in fact, in some cases, they're so damn sick of it in like machine learning, someone actually described – told me, he said, "No, we're in our post L1 norm stage," meaning that they're so sick of it and everybody's done it, and they've already had their 300 papers on it at the last Nips or whatever it was. And so they're actually just sort of moving beyond it, which is fine.

Okay. So and I'll say more about the history as we kind of get into it. So, and it's got different – oh, the other thing to warn you about is it's got different names. I don't know if they're even in the slides here. The names, you're gonna hear about it, are – some people just call them L1, L1 norm tricks. Let's see. There's another – you'll find in statistics you get lots of names. I don't know why. You get lasso. Let's see, basis pursuit, several others.

Oh, in machine learning, you get things like SVM is support vector machine. People who do this will never admit that these are just L1 tricks. Trust me, they are. So they'll never – yes, and I hope some of you out there are watching this online and getting pissed off. So that's exactly why I said it if you're wondering, because I don't – there's no one to piss off here.

Anyway, so support vector machines, and let's see. Oh, it's used in actually geophysics actually I think is one of the earliest examples I can find. That's right from Stanford.

That’s 70’s. That’s [inaudible]. And there’s a rumor of, and I’ve heard something about some papers on antenna design from the 40s. I’m gonna – I have to hunt those down using these methods, so okay, all right.

So let’s go over this. These are – they all involve problems with cardinality. So now cardinality is just the number of nonzeros. Well, of course, it’s the size of the set, but the cardinality function on a vector is simply the number of nonzeros in it. So these come up a fair amount. They’re actually hard to solve exactly. A lot of them are MP hard. Some of them, people have shown that, others not. Some might be easy, but there’s a very limited number that are easy, the simple mapsac problems and things like that.

But the vast majority of these things are hard.

Okay, so the other interesting thing about it is that it’s simply [inaudible]. It relies on the L1 norm. Oh, I forgot to mention my list of fancy names. Compressed sensing is one that’s making the rounds in statistics and applied mathematics, so you’ll hear that. And maybe there’s at least one or two other – it’s got lots of names, but as I think of them. Oh, here’s some right here, okay.

So it’s been used for a long time. In engineering design, it’s been used for a while, and we’ll see where that comes up. It’s been used in statistics. Again, there’s a Stanford connection because the originators of this are at Stanford. Let’s see. Oh, it’s also related to [inaudible] that goes in the 70s. Support vector machine and machine learning. Oh, this is from the maybe 80s or even maybe early 80s, this total variation reconstruction signal processing and geophysics.

This again, like support vector machine, do not, if you see a person who does total variation reconstruction, don’t say, “Yeah, that.” Actually, do say, “Oh, yeah, yeah, sure. That’s just an L1 trick,” because it will really piss them off. Okay, and compress sensing, that’s the newest import. Okay, there are some new theoretical results. They are very cool because it really didn’t occur to anyone that you could actually say anything about these methods. Those are very cool. I’ll mention them, but needless to say, that’s not gonna be our focus.

Okay, so cardinality of vector, it’s actually a separable function. So it’s a sum of – it’s this. It’s actually the sum over all XI's of card of XI. But card of XI is nothing but zero or one, depending on whether the variable is nonzero or not. That’s quasi concave on R plus to the N, but not RN, for what that’s worth, which is not much because that would tell you that if you wanted to maximize the number of nonzeros, you could do it of some positive things, but that’s kind of silly.

Otherwise there’s really no convexity properties, and it arises in lots of problems. And let me just – I mean I know this is kind of obvious, but I’m just gonna draw the picture of the function just so we all know what it looks like. It looks like this. Here’s X, and it looks like this. It’s that and then that. So this is pretty non-convex, I think you’d have to say. Actually, we'll leave that picture up because it's gonna – the whole thing's gonna be highly implausible, I mean why these things work.

Okay, so convex-cardinality problem, it's a problem that would be convex, except cardinality comes into the problem somewhere. And it might come in the objective, but it also could come in a constraint. And these are really interesting. This says – these are very interesting problems, so is this one. This says the following. This says, "Find me the sparsest vector that lies in a convex set." Boy, are there a lot of applications of that.

For example, C could be the set of coefficient vectors that explain your data within some reasonable level of confidence, okay. So these are the – then this says, "Fine me the simplest description explanation. Find me the simplest model." So that's what this is. So this is even – that would be like [inaudible] or something like that.

It comes up in engineering design as well. So here, for example, the Xs are often things like widths or something of wires, let's say in a circuit. And then if a wire has zero width, it means the wire isn't there. So it's basically saying, "Make me a power supply distribution network or something, or a clock distribution network. Obviously, you can't give a tree because if you get a tree, you can't remove any wire segment, so you get a mesh.

And then you're basically saying, "Find me the sparsest thing that will actually this thing that will actually satisfy all the specifications, okay. This comes up lots of places. Another good example would be design me an FIR filter, and these are the coefficients in the filter.

What's cool is if I have a filter and it only has something like 14 nonzero coefficients, zero coefficients, I don't have to – I could build custom hardware that will just have 14 multipliers, I mean, or that could be base or time, but you get the idea. You can make very, very – as far as FIR filters, you get implemented hardware very quickly. Actually, software as well if you do it the right way. So all right, it comes up in lots of cases. We'll talk about it in all of them.

This one says, "Choose me no more than K of these coefficients." Okay, so how do you solve these problems? Well, the first thing is this. If you fix the sparcity pattern of X, okay, of which there are two to the N. So to fix a sparcity pattern says that for each component, it's either nonzero or zero. There's two to the N choices like that. One choice is, of course, X equals zero. The other choice is X is full, so all things are there.

There's two to the N sparcity patterns, but if you fix a sparcity pattern, you get a convex problem and it's all over. Now what this says is that if N is, let's say, ten, these cardinality problems can be solved immediately and globally because it just solved one or two for convex problems and you're done. Everybody see what I'm saying?

And, in fact, there's tricks there, you can even do it faster than that if you know what you're doing because, for example, you can do things like this. Once here, suppose we want to find the – minimize the cardinality. The sparsest X is in some set. This might be some design. Once I have found, if I have ever found an X with a cardinality, say, of eight, I never have to look at – obviously there'd be no reason for me to look at any sparcity pattern that has more than eight, right.

So you can often trim a large number of these things, even if you're doing direct enumeration. But you can simply do direct enumeration immediately if there are, for example, just ten variables, or 20. Twenty is a million. That's overnight or something, or in C, it could be some reasonable amount of time. Okay, all right.

Now the general problem is NP hard. That's actually quite easy to show, and it's not that relevant either. Now you can solve these methods globally by branch and bound method. We're gonna do that later in the class. These are global optimization methods, and these things, well, they can work for a particular problem. And these are gonna be – this is gonna be the global solution.

But in the worst case, these things reduce to checking all of the sparcity patterns, or at least many of them or something like that. So these methods are actually heuristic. So there's a very good way to write these out as a Boolean LP, a mixed – and there's a way to write these out. So a Boolean LP looks like this. You minimize a linear function subject to linear inequalities. But these Xs are simply zero one.

Now this problem here, you can embed essentially any combinatorial problem as a Boolean LP, any with a reasonable embedding. I mean not one that grows size or anything like that. So you take your favorite three traveling salesmen, and I'll easily write it out this way. Now if you want to see how to do it, it's actually pretty straightforward. What you do is this, is you write down this. You write minimize C transpose X, X less than B. And then you write the cardinality of X plus the cardinality of one minus X is less than N, okay, less than or equal to N.

Now what' interesting about that is the only way that will work is that the XI's either, for each entry, either this one is zero, either that zero or that zero, and that's the same as saying X is either zero or one. So it's a very quick embedding, okay. I mean not that it matters much.

Okay, so now we're gonna talk about applications. I already hinted at this, but here's one. Oh, and by the way, I think these methods have not diffused anywhere near the extent to which they should have, right, because what's happening is a lot of work on this. You know, some people know it, but they know it in a weird way. Like if they're in statistics, they just do SVM's or something or L1 regularize this or that.

They haven't really internalized this. It's just a completely general heuristic, powerful heuristic for getting sparse things period. And there's lots of people poking around who could use these things and have no idea about these matters. They go to Google.

The next thing they know, they're reading some fantastically complicated math paper with a result saying that if you do this in some incredibly special case and something's bigger than something else, then you'll actually get the global minimum with overwhelming probability or something like that. And that's not – what they needed in fact was simply an excellent heuristic for solving sparse problems or something like that. So anyway, all right.

The sparse design is this. This is the set of vectors that satisfy a bunch of specifications, and you want the sparsest design and so you can have FIR filters. Like I said, one of the first applications, this was antenna array beamforming. And here the zero coefficients correspond to the unneeded antenna elements.

So truss design is another one that's actually very – this is actually not only – this is actually now has huge commercial impact, so there's a whole industry that does this kind of stuff. And it's not just trusses, but also to mechanical stuff. And here's zero [inaudible] bars that are not needed. Wire sizing, I already mentioned that.

Now what's interesting about these applications is, for example, let's take the antenna array design thing. So what you do is you start with you say, "Here's where I'm gonna put my antennas on something." And you start with 1,000, and someone says, "Well, you can't possibly put 1,000 antenna elements there," because first of all, we can't afford the – you can't afford that many. They won't even fit physically. You can't even fit them all there.

The second is you have to have an RF amplifier for each one and blah, blah, blah. It goes on and on. And then you say, "No, no, no, no, these thousand antenna array elements, you shouldn't think of them as possible antennas." I have absolutely no intention of using 1,000. In fact, I intend to use, let's say, 17. So then you say, "Well, then why are you using 1,000?" And you say, "Ah, because this method is gonna pick which 17 I'm gonna use."

And the same in truss design. You make something and you put 10,000 trusses or 20,000 bars in the truss. You have no idea – you have no intention of building a truss that's 20,000 bars or 200,000 bars. You're gonna let this thing decide on the 132 bars it's gonna use, and that's a reasonable truss.

So okay, so this is kind of obvious. In statistics and modeling, it's completely pervasive, absolutely pervasive. I mean, and it's completely fundamental as well. It basically says the following. Oh, this would be a simple one. This says, "Choose K columns of A to get the best fit with B." So, and that's a cape term model, and again, here you could do this by simply enumerating all N choose K choices. This is quickly gets out of hand if K is more than a handful and is big and so on and so forth.

The variations would be find me the – choose me the minimum number of columns which allows me to get a good fit. You could put it this way. I mean there's lots of variations. This is progressive selection. By the way, this could be quite interesting because the columns of A here actually are regressors or features. And so you're actually doing – this is in other people and say, "This is what we call the machine. We even cause feature selection."

So this would be feature. And you could throw in all – so what this means is actually kind of turns a basic principle in modeling on its head. So let me exaggerate here, but say that basically as of ten years ago or before these methods were proliferated, the basic story in modeling goes like this. You say, "I want to predict the next term in this sequence. I have this sequence here." And someone says, "Fine."

You construct a model, and you construct something like autoregressive model. And you say, "I'm gonna make a linear combination of the last ten entries or something," or it could be the last ten entries plus weird quadratic terms plus sick things that your domain experts throw in there, like they say, "Oh, here's an interesting feature. How about this?" The highest value in the last five steps, you know. Who's to say? How about the spread? The largest minus the smallest value? How about some weird time weighted maximum? I mean you can go nuts.

By the way, this is all good from an employment point of view for the experts, right, because if it takes the experts to generate all these features, that's great. But then you say, "No, no, wait, wait." Oh, obviously the more features you add, the better you can fit your data. So if I give you a pile of data and if I make – if I give you 100 data points and I have like 100 features, then generically, I'll be able to fit it exactly and say, "Wow, I'm doing really well."

That's stupid. It will have no generalization ability, and you would verify that by cross validation. So you'd take your data, have two sets. You could even be formal about it, not even allow the person developing the model even access to the other one, and then you'd try out this way and you'd find over fit. So the problem is you need the expert to choose a small number of features. Everybody following this story?

So this is modeling, statistics, whatever you want to call it as of 15 years ago. The story is everything comes down to choosing the right features. You get the right – and a small number of them too, by the way. So notice that puts the burden on the modeler, so that's the idea.

And then how would they do it? They'd sit around at home or whatever. They'd actually have their grad students do it more likely. And they would try various combinations of things, and someone would say, "Oh, my God. I got an amazing fit." It turns out that the volume minus the maximum minus the spread, blah, blah, blah. If I throw in these six features, I get an extremely good prediction of this. Everybody see what we're saying here?

Okay, all right. So this completely turns it on its head. This basically says you can do feature selection, and what's super cool about this is it could be fat. If you have a fat A matrix in a general regression problem, that's just stupid. That's you're automatically over fit. You're way over fit. There'll be lots of combinations that will reconstruct your data with no – zero error. They will have no generalization ability, but so that is totally uncool in a fitting problem, okay.

So now here it works fine. What you do is you get – you have to have something like 100 measurements, and you give 10,000 features. So not only are you doing a bad thing, but you're doing it by a factor of 100 to 1. Everybody see what I'm saying? You're so over fit, it's ridiculous.

But what you do is you solve this problem and you either put a cardinality or you solve one of these problems here. And what those 10,000 features are, you have no intention to use 10,000 features to fit your 100 data points because you could fit them in a huge 9,900 dimensional null space of ways. You could fit them exactly.

So what you're actually saying here is the following. These are 10,000 features. Most of them are probably completely irrelevant, and in any case, I'm not authorizing you to use 10,000. You may choose 30 of the 10,000 features, and that's exactly what this problem would do. So by the way, the implications of this, I think are gonna ripple for the next ten years. It's gonna completely change modeling and a lot of other stuff. Everybody see what I'm saying here?

So this is very – I mean normally people talk about this is a much lower level or everybody knows all these things, but I actually haven't heard people say it this explicitly. But everybody got this here? So you can do wild. I mean so, by the way, it's interesting. You got to totally change your thinking. If someone says, for example, "I want to fit this data with a polynomial of these things," and someone would say, "Well, that's a pain in the ass, and it's not gonna work well," because if you have ten variables you want to fit a polynomial, linear is fine. That's ten coefficients.

Then you go to quadratic. How many coefficients you got? Come on, a quadratic and ten variables, how many coefficients? I always have two answers, one of which is wrong, but not – it's only off by a factor of two. It's about 50, right? You have 50 coefficients in it, and then you go, "Well, how about cubic?"

Well, then you get 1,000 divided by three. You get 160 or something like that in a cubic. And so the problem is if you have limited data, when you do polynomials, if you do generic polynomial fitting, you just get piles and piles of coefficients. Everybody see what I'm saying here?

Okay, so this sort of changes that because you could do these methods with sparcity. You can fit a polynomial. It will be a nice sparse polynomial. It might turn out that some sick polynomial degree four that this thing picks out that works really well. So okay, I think I said enough about that.

Sparse signal reconstruction, this comes up, I guess as people doing it. There we go. There's one. There's another one though. There, there you go. There's two at least. Okay.

So sparse signal reconstruction goes like this. It says estimated signal. You're given some noisy measurement, and you're given the information that X is sparse, okay. That's like prior information you're given. That's – so that's what you're given.

And that comes up in a bunch of cases, like it could be false anomalies, which I guess it is in your case. What is your – oh, it's sparse coefficients in the radon transformer. It's gonna come up a lot. Okay, and the – if you want to do maximum likely estimate, and I tell you that something has only K nonzeros, you would end up minimizing the subject to that. I mean you can't solve this problem, so but you're gonna do an approximation of it.

So and this is actually very important. You should think about this, about sparse [inaudible]. It's very, very interesting to think about, and I'll give you an example. If you go back to sort of the last – if you go back to sort of the last century, everything is based on kind of lease squares. And if someone – if there's a statistician who says, "Justify that," you would say Gaucian or something, and then that would turn into some now statistically defensible reason why you're using lease squares, right? That's how these things work.

But sparcity doesn't really come up, so sparcity would come up and once you start thinking about it, you start realizing, well, there actually are lots of cases where a single model is that it's sparse. One would be like set commands or something, set point commands or something like that. If you build some sort of a controller for this based on all the standard stuff, you're really saying that the input is some Gaucian process.

But in fact, what you really might think or want to say is something like this. No, the input actually is constant for long periods of time and then switches to another value. That's actually what – that's the way set points and commands actually look. That's sparcity, so that would be that kind of thing.

Okay, so we'll – okay. Estimation with outliers is another beautiful example, and it works like this. You have YI is AO transposed X plus V. This is just a linear one to make it simple. The VI's are our friends that were with us last century. Those are our nice Gaucian noises, our catchall. It's basically this is our statistical excuse for using lead squares. That's what this means, okay.

But then you have this other weird noise, W, and the only thing you assume about it is that it's sparse. So, for example, it has fewer than K entries or something like that. And basically what these are is these represent either anomalies, outliers, or bad measurements, so – and this comes up a lot.

So basically this says that every 100 samples, the measurement you get is just completed unrelated because you don't assume anything else about W. And, of course, by choice of this thing, I can make this thing anything I like. So it's something that's complete – just simply unrelated to the input, but you're not told that. So you're not told that. You're simply told, "I'll just give you the measurements."

By the way, if these are all on the order of like – on the order of one and W is like ten to the five, this is not a hard problem because the measurement comes in. They've all been coming in around plus or minus one, and one comes in as ten to the five. It's a good guess that's an outlier. Anybody can handle that, okay. The really insidious, cool way is this, is when the bad measurements come in, they have the same statistics as the good ones.

So when you have a measurement of .8, whatever is plausible, and the completely bad measurement comes around, and it's .8, that's much trickier to do. Actually, you'll find out these methods that we're gonna look at here, they'll handle that like amazingly well, I mean amazingly well where I don't know of any other method really that would sort of get close to doing this.

Okay, anyway, so you get this. You're basically doing – you're estimating the set of outliers and so on, and it would look like a problem like that. By the way, you'll see that when we replace this with an L1 norm later, this is gonna recover robust statistical methods that have been used for 25 years in statistics, so okay.

Here's a very interesting one. Suppose you have a set of convex inequalities, and they're not feasible. They're not mutually feasible. So, for example, let's do this. Let's make these a bunch of design requirements. Oh, let's make it a circuit. So this is a power limit, an area limit, a timing constraint or something like that. They're not mutually feasible. They cannot be satisfied, so the intersection is empty. Then you'd say this.

You'd say, "Well, first of all, that's good to know," because you'd go back to the person and say, "Your specifications are infeasible." And they'd go, "Well, you can't do it. Maybe I need another designer." And you go, "Oh, no, wrong. This problem is convex. And when I say it's infeasible, I don't mean I failed. I mean everyone will fail to do it." And they go, "Well, uh huh, fine." And then they go, "That doesn't help me. Of the 17 constraints, please come back with a design that violates as few as possible." So that's how you could do it.

And in fact, what you could do is you could classify your constraints into the ones that are non-negotiable. And you could then have the other ones that you really wanted, but can't have. And for that latter group, you say, "Okay, of these 10 or 50." Oh, if I said ten, it's no problem because two to the ten is 1024, and that you can solve immediately, so I should have said, "Of these 100 inequalities here that I'd like to have, please find me a solution that violates as few as possible." Everybody see what I'm saying here?

Okay, all right. So you would do this. You'd say, "Minimize the cardinality of a vector T subject to FI of X is less than T, and T is positive." Now what happens here is this, is whenever T is zero, it means that the Ith inequality is actually satisfied. If you can't satisfy the Ith inequality, T is sort of like a little fudge factor. So T positive means you've been given some slack here. So this thing here will immediately and directly solve the question of finding the minimum number of violations.

By the way, these are really useful. By the way, we'll see also, you've already seen this, you've used it, we've talked about it before. It's an L1 phase. It's a sum of violations phase one method. So, but it's good to see all of these in the same context. Here's one, linear classifier with the fewest errors. So we'll do some machine learning type stuff. So I have a bunch of data. I have X1, Y1, XM, YM. These are vectors, and then these are binary labels, so they're plus minus one.

And what I would like is I would like to find a vector W and an offset V so that sine of W transpose X plus V is Y. Now approximately equal means something like you make as few errors as possible or something like that. You do it right if YI times this thing, actually, if YI times – if the sine of this – well, if you make this positive, that means doing it correctly, if these have opposite signs, that means you've messed up.

So if you want to find the W and V that give the fewest classification errors, that's basically you have a bunch of things that are labeled. If you can get zero classification errors, that means you can put a hyperplane between the points, but now you want to put the hyperplane there that has the fewest number of things on the wrong side, and that would just be this problem here.

Now I should remind you something here, and that is that not one of these problems that we're writing down can be solved, so that's important to understand. They're all hard, so you might ask why are we writing these down, right. Oh, we can solve them in some cases, right. If there's ten – if I've got ten weights, I can solve this, okay. If this is ten features, I can solve this problem by solving 1,024, sorry, yeah. No, sorry, if M is ten, I can solve 1,024 convex problems, and I will get absolutely the linear classifier that has the fewest number of misclassifications. But if M is bigger than that, you can forget it.

We're writing these down this way because later we're gonna see a heuristic that's gonna apply to all of them, and you're gonna get good methods.

Student: Why is there a one?

Instructor (Stephen Boyd):Yeah, why? Because to make it right is to make this positive, to make this thing positive. Yeah, so you can't write down a strict inequality in a problem. Well, you can write it down all you like. It won't do the right thing. You need strict positivity.

However, this – that constraint, when I make his strictly positive here, is homogeneous scaling WNV, and therefore, to say that it is positive is exactly equivalent to saying that it's bigger than or equal to one by scaling WNV. So this is a standard trick for that. Homogeneous problem with a strict inequality, you kind of replace it with a one.

Okay, here's a variation that's actually extremely interesting. This one is a variation on finding a point that violates as few as possible. In fact, it's the dual, and it works like this. I give you a set of inequalities, which are mutually infeasible, and what I want to do is the following. If someone says – if you said, "Well, I have 28 specs, let's say on phase, margin, power, rise time, slough rate, blah, blah, blah."

You'd say, "I have 28 of them. They're mutually infeasible. No one can design that amplifier period." So then you say, "Okay." The question then is what's messing you up. So and let me – first, let's figure out what would be a powerful statement. A powerful statement, if I gave you 28 inequalities, would be to say, "Well, actually, as it happens, inequality number 14 alone is infeasible."

That's a very powerful statement, right, because it means that basically you have to deal with that one. If you're gonna relax one, you have to relax that one. Everybody see what I'm saying? A slightly less powerful statement would be, "Well, 14, 22, and 37, those three together are mutually infeasible."

And the weakest statement I could make is that they're all mutually infeasible, but if I remove any one inequality, they become feasible again. Everybody see what I'm saying here? That's the weakest statement you can make. So basically the smaller the cardinality of a subset of infeasible inequalities, the stronger the statement, the better you have identified what the problem is, what's causing the infeasibility, okay.

So how do you do that? Well, you do it as follows. You look at the dual function. That's G of lambda. That's this thing. That's again by homogeneity. It has to be bigger than one and lambda positive. And so what you'd do is you'd minimize the cardinality of lambda subject to G of lambda bigger than one, and lambda bigger than or equal to zero here.

If lambda is zero, it means that you – if lambda I is zero, it means that you didn't use FI in constructing your proof of infeasibility. And it means – so a sparse dual certificate basically corresponds to identifying a small subset of mutually infeasible constraints, so that's how that works.

Okay, we'll do another one. It is portfolio investment with linear and fixed costs, so let's see how that works. We're gonna – we'll take long positions, so we're gonna invest an amount, XI, and they're gonna sum up to be here, but we'll get to that in a minute. No, that's our budget. Sorry, they're not gonna come up to B because we're gonna have to buy.

We're gonna actually gonna have to – we're gonna purchase these things, and the way that's gonna work is this. So here I'm gonna have a trading fee, which is sort of alpha. This could just be a scalar. It doesn't really – well, sorry, a scalar times one. If I buy an amount, I'll charge you something like 1 percent, but also if you buy a stock at all, if you initiate a trade, I'll charge you $50 bucks plus 1 percent. That would be for like a sad consumer like us, or something like or something like that, right?

So that's the way this would work. So the budget constraints is this. Let's see. That's the total amount of the stock that you – that's the value of the stock you purchase, that's your trading cost, and that's less than your budget. That's it. And you just say, "We'll just do a simple 1952 mean variance problem." So your mean return on the investment is miu transpose X, and the variance is X transpose sigma X. And so you might say something like this. "I want a minimum return of R min, and I would like you to minimize the variance, the risk of this portfolio, okay."

And then this is the constraint that says that you're sort of within budget. Now, of course, when you do this, this will be equals here, okay. So that's a relaxation. It doesn't matter because you can't solve that because of this thing here. Now this one is actually quite interesting.

What it says is well, let's even just talk about it intuitively for a bit. If you invested in one stock, then your fixed trading fee is gonna be very small. It's just gonna be beta. So you're only gonna have lost, let's say, $50 bucks if you invest in one. Now the problem with that is you may – you're gonna get high risk. The whole point investing in a couple of them is to have the risk go down.

So if you have a couple of assets with a correlation coefficient that's away from one, I mean ideally it would be negative, in which case you can hedge, but that wouldn't actually happen. But let's suppose they were a reasonable correlation coefficient was not .9 or .8. If you invest in two, your risk goes down. The problem is if you invest in two, you're paying $100.00 in fixed fees.

So this is kind of the idea. If you solve it without – if you ignore – if you say beta equals to zero, that's a convex problem, and you can actually find out. That'll generally, by the way, invest in a bunch of different things, okay.

Let's do piecewise constant pitting. So here you have a fit, you want to fit a corrupted signal, X corrupted, by piecewise constant signal, X hat with K or fewer jumps. So in other words, I have some measurements of – I have a signal, which I believe to be piecewise constant and to make a jump every so often. And over my time of interest, I believe it's gonna make K jumps, but no more, okay.

I can give you very noisy samples of it, and you want to fit it with a signal that's piecewise constant. All you have to do now is this. You form a difference matrix, and DX hat is actually the difference vector, and the cardinality of that is the number of jumps. So if it's N minus one, it means that at every step, X hat changes. If it's zero, it means that X hat is constant, it's flat.

Okay, so your problem might be this. Get as close as you can to the corrupted signal with a piecewise constant signal that has K jumps. By the way, some of these you could actually solve. This one you could probably actually solve. That one's probably not hard.

You could also do piecewise linear fitting. So piecewise linear fitting, you do the same thing, but you take a double difference operator, so a del square operator or something here. So you take the second difference because this thing gives you the second difference. And a jump, if del X hat is zero, it means that X hat is the average of its two neighbors, and that means there’s no curvature there. There’s no curvature. If this is sparse, it says that the signal X has piecewise linear. So this is a piecewise linear signal with K jumps, so okay.

Finally, we’ll get to the L1 norm heuristic. It’s embarrassingly simple. The simplest version of it, we’ll look at much more sophisticated ones later, but the embarrassing one is this. You simply replace the cardinality with some multiple of the one norm. It’s that simple, or you add it to the objective, and then gamma is a parameter that you use to tune things.

Now we’re also gonna see weighted ones like this. But actually, before we go on to that, I want to just draw the picture over here just to show you how completely ridiculous this is. So here it is. Here’s the actual function. Here’s the cardinality function. I’ll just draw it like that so you can get the idea. There’s the cardinality function.

And this heuristic says, “Approximate this function with an absolute value.” So I don’t think there’s no one. There are very few people that would argue that this is a good approximation of that. I mean that takes some serious storytelling abilities to convince someone that that’s a good approximation of that. And, in fact, when you look at this picture, because that’s all you’re doing. You’re placing a function that looks like this.

It’s one everywhere. It drops to zero right at zero. You’re replacing that with an absolute value. So the whole thing looks highly suspect, highly implausible, doesn’t make any sense, but that’s it. Remember the same thing goes with La Grange duality. I don’t know if you remember that. You approximate the – your irritation function that looks like this with slope, so – and that worked out. In here, it’s not gonna work out. It’ll be a heuristic.

Okay, so let’s just do an example here. We start with this hard problem. Find me the sparsest X that’s in some convex set. The heuristic says minimize the one norm of X subject to X in the convex set. This is convex. That’s an easy problem to solve, and what can you say about it?

The solution of this is not the solution of that. We’ll see what you can say, there’s cases where you can get a bound on this solution from that, and there’s a very, very small number of cases, extremely specially. Basically C is an alpine set, so it’s just a quality constraint in which there’s actually a statement you can make.

However – I mean a statement that’s actually true. You can always say this. This seems to work unusually well as a heuristic, this and variations. So as a cardinality constraint problem you could do many things. You could add this this way and then sweep beta. You could also add it in here. And you would adjust these numbers so that the cardinality is less than K.

And I think I’ll mention one quick thing. I think we’ll do one example and then we’ll quit. So polishing is quite interesting, and let me explain what that is. It’s not universally used. Statisticians don’t do it for some reason, and they tried to explain it to me and I didn’t get it. Apparently it’s bad, but anyway, at least in the context of their problems. But for other applications, it’s not.

It works like this. You use the L1 heuristic to find an X hat with required sparcity. Now once you’ve done that, you fix the sparcity pattern, you go back and you solve the problem again. That’s how that works, and that’s called polishing. That’s not a standard word. I just made it up, but it sounds like a good word, so a good description of it. Are you doing that?

Student: Not yet.

Instructor (Stephen Boyd):You will now.

Student: Yeah.

Instructor (Stephen Boyd):Okay, good, so good. Yeah, it doesn’t take much, okay, so to do this. And I think what we’ll do is maybe we’ll quit here. Actually, what I would like to do is just to get to a quick example, and then we’ll come back just for fun. There we go, okay. So we’re gonna do a quick example and then just so I showed you one.

So we want to find – this is the regression of selection problem. We want to select columns of A to fit B with, K of them. And so what we’ll do is we’ll replace this. We’ll do an L1 thing. You actually – you add this plus lambda times the one norm of X, and you trade it off. This is a tiny little problem, although it’s already big. I could actually do this if I wanted to because two to the 20 is about a million.

And so this one, I can actually find the global solution. In fact, I believe this is the global solution. This is the global solution calculated with a million – by solving a million lead squares problems. What this shows, the solid curve is what happens if you do the L1 heuristic with polishing, and you can see it works pretty well. Everyone agrees sort of here, but what happens now is – and you can see any gap between these two is where you’ve missed the global optimum.

What’s interesting is, for example, over here with three, you actually get the global optimum. In other places over here, one, you get the global optimum. I guess you nailed it at six as well, but notice that you’re never really very far off. So and considering that the effort in calculating this curve and this curve in this case is off by – is a factor of ten to the six. You’d have to say it’s not bad. I mean this is not bad. So for solving at a million times faster, you get close.

So okay, so this is a good time to stop, and then we’ll continue next time. And oh, before you leave, I just – there is a tape ahead tomorrow, 12:50 to 2:05. It’s on the course website. We’ll quit here.

[End of Audio]

Duration: 78 minutes

ConvexOptimizationII-Lecture15

Instructor (Stephen Boyd):All right, I think this means we are on. There is no good way in this room to know if you are – when the lecture starts. Okay, well, we are down to a skeleton crew here, mostly because it's too hot outside. So we'll continue with L_1 methods today. So last time we saw the basic idea. The most – the simplest idea is this. If you want to minimize the cardinality of X, find the sparsest vector X that's in a convex set, the simplest heuristic – and actually, today, we'll see lots of variations on it that are more sophisticated. But the simplest one, by far, is simply to minimize the one norm of X subject to X and Z.

By the way, all of the thousands of people working on L_1, this is all they know. So the things we are going to talk about today, basically most people don't know. All right. We looked at that. Last time we looked at polishing, and now I want to interpret this – I want to justify this L_1 norm heuristic. So here is one. We can turn this – we can interpret this as a relaxation of – we can make this a relaxation of a Boolean convex problem. So what we do is this. I am going to rewrite this cardinality problem this way. I am going to introduce some Boolean variables Z. And these are basically indicators that tell you whether or not each component is either zero or nonzero.

And I'll enforce it this way. I'll say that the absolute value is XI is less than RZI. Now R is some number that bounds, for example – like it could be just basically a bounding box for C, or it can be naturally part of the constraints. It really doesn't matter. The point is that any feasible point here has an infinity norm less than R. If we do this like this, we end up with this problem. This problem is a Boolean convex. And what that means is that it is – everything is convex, and the variables, that's X and Z, except for one minor problem, and that is that these are 01. Okay? So this is a Boolean convex problem. And it's absolutely equivalent to this one.

It is just as hard, of course. So we are going to do the standard relaxation is if you have a 01 – 0, 1 variable, we'll change it into a left bracket 0,1 right bracket variable. And that means that it's a continuous variable. This is a relaxation. And here, we have simply – we have actually worked out, this is simply – well, it's obvious enough, but this is simply the convex hull of the Boolean points here. Now if you stare at this long enough, you realize something. You have seen this before. This is precisely the linear program that defines – this is exactly the linear program that defines the L_1 norm.

So here, for example, this is at norm X is – it's an upper bound on – ZI is an upper bound on one over RXI. And so, in fact, this problem is absolutely the same as this one. And so now you see what you have. That Boolean problem is equivalent to this. By the way, this tells you something. It says that when you solve that L_1 problem, not only do you have – is it a heuristic for solving the hard Boolean problem, it's says it's a relaxation, and you get a bound.

The bound is this: you have to put a one over R here, where R is an upper bound on the absolute value of any entry in C, or for that matter any entry that might – is a potential solution or something like that. And this tells you that when you solve this L_1 problem not only do you get – is it a heuristic for getting a sparse X, but in fact it gets you within a factor of R a lower bound on the sparsity. So that's that. By the way, it's a pretty crappy lower bound in general, but nevertheless it's a lower bound. Okay. Now we can also interpret this in terms of a convex envelope. And let me explain what that is.

If you have a function, F, on a set C – and we'll assume C is convex, so on a convex set C – in fact, I should probably change this to convex set. This function is not convex. The envelope of it, it is the largest convex function that is an under-estimator of F on C. And let me just draw a picture, and we'll – I'll show you how this works for the function we are interested in. The function that we are interested in looks like this: it's zero here, and then it's one over here. So that's our indicator function. And, if you like, we can do this on the interval, plus one minus one, okay. And so our function looks like this, basically.

That's our cardinality function. And what we want to know is this. What is the smallest – sorry, the largest convex function that fits everywhere beneath this function on that interval? By the way, well, I can leave it this way. So the simplest – any convex functions, it's got to go through this point; it's got to go through that point; and therefore, this gives you – that's it. So no one would call – no one would call this absolute value function a good approximation of this function, for sure. But it does happen to be the largest convex function that's an under-estimator of it, okay. So that's that.

And then, actually, you can go like this, if you like. Because it's on this – it's just on this one set. So that's the convex envelope. Now we can relate to all sorts of interesting things. I mean one is this that the – one way to talk about the envelope of a function in terms of sets is this. You form the epigraph of the function, and then simply take the convex hull. And it turns out that's the epigraph of the envelope. And you can see that over here, too. So the original function has an epigraph that looks like this. It's all of this stuff, and then this little one tendril that sticks out down there. So it's everything up here, and one little line segment sticks out there.

Convex hull of that fills in this part and this part. And what you end up with is the absolute value restricted to plus minus one. So that's going to be the convex envelope here. And another interesting way to say it is this, it is, in general, it's F star star. And I don't want to get into technical conditions, but – actually, the conditions aren't that big of deal, it's just that it should be closed or something like that. Actually, this function here is not closed so – but anyway, it looks like this. It's the conjugate of the function, which is always convex, and then that starred. So it's the conjugate of the conjugate.

So now if the function F were closed and convex originally, this would cover F. That's kind of obvious. Okay. So for X scalar, absolute value of X is the convex envelope of card X and minus one one. And if you have a box of size R, an L infinity box here, then one over R norm X1 is the convex envelope of card X. So that gives you another interpretation. So if someone says, "What are you doing?" You say, "I am minimizing the L_1 norm in place of the cardinality." They can say, "Why?" You would say, "Well, it's a heuristic for getting something sparse." And they go, "Well, that's not very good. Can you actually say anything about it?"

And you go, "Actually, I can. One over R times my optimal value is a lower bound on the number of nonzeroes that any solution can have." Okay, by the way, if you understand this, you already know something that most of the many thousands of people working on L_1 don't know. You are actually now much more sophisticated. And I'll show you, and it's going to be stupid, but it's actually completely correct. Suppose for some reason I told you that X1, Y is between one and two, so it lies here. What is the convex envelope of that? Well, it's this. It looks like that. And what you see is that the function is now no longer an absolute value.

It's asymmetric, okay. So it's asymmetric. You can write out what it is. Would it be a better thing to do if you wanted to minimize the cardinality? In this case, the answer is absolutely it would be better. It would work better in practice. In theory, of course, it's better because it gives you the actual convex envelope and so on. So what that says is that when you minimize the one norm of X as a surrogate for minimizing cardinality, you are actually making an implicit assumption. The implicit assumption is that the bounding box of X – of your set – that the bounding box is kind of – it's like a box. It's a uniform, right.

All of the edges are about the same, and they are centered. If you ever had a problem where you are minimizing cardinality and X is not centered, like for example, some entries lie between other numbers, you would be using a weird, skewed, weighted thing like that where you have different positive and negative values. Oh, what if I told you that X2 lies between, for example, between two and five? What can you say then? Then X2 is not a problem because X2 will never be zero. So it's just – it's a non – then it's easy. Okay. Well, we just talked about this.

If you had a – if you knew a bounding box like this with an LI and a UI, then – and by the way, you can find bounding box values very easily by simply maximizing and minimizing XI subject to over this set. So you can always calculate bounding box values. Now if the upper bound is negative or the lower bound is positive, then that's stupid because that means that X has a certain sign, and there is no issue there. It is a non-issue. If they straddle zero, that means there is the possibility that that X is zero. And in that case the correct thing to do is to minimize this. If these things are equal here, that reduces to L_1 norm minimization.

So that's what that is. This will also give you a lower bound on the cardinality. Okay. So let's look at some examples. I think we looked at this last time briefly. I'll go over it a little bit better this time, or we'll go over it in a little bit more detail. This is a regressor selection problem. You want to match B with some columns of A, linear combinations of columns of A, except I am telling you, you can only use as many as K columns here. So, okay, so the heuristic would be to add, for example, a one norm here and adjust lambda until K has fewer than K nonzeroes.

And then what would happen is – you'd look at this value of that then. So here is the – here is sort of a picture of a problem. It's got 20 variables, 2 to the 20 is around a million. And therefore, you can actually calculate the global solution. You can do it by branch and bound. We are going to cover that later in the quarter, but you can also – in this case, you just work out all million. So one million least squares problems, you'd check all possible patterns, and not a million. Yeah, yeah, this is a million. You just solved a million of them and for each one. So the global optimum is given here, like this.

And this one gives you this – the one obtained by the heuristic. And you can see a couple of things here. It looks to me like you never – I am not quite sure here, but I think for most of it you are never really off by – well, no, here you are off by two. That's a substantial error. You are off by one; sometimes you are exactly on and stuff like that. But the point is that this curve is obtained by the heuristic, which was one-millionth the effort there. Okay. So now we'll look at sparse signal reconstruction. It's actually the same problem, different interpretation. You want to minimize norm AX minus Y.

Y is a received signal. X is the signal you want to estimate. The fact that there is a two norm here, this might be, for example, that you are doing maximum likelihood estimation of X with a Gaussian noise on your measurement, so you have your Y equals X plus V. Then this is prior information that the X you are looking for has no more than K nonzeroes. So that's the – that's the other. The other one, the heuristic would be to minimize this two norm subject to norm X in one less than beta. In statistics, this is called LASSO here. I can't pronounce it – you have to pronounce it with Trevor Hasty's charming South African accent.

I tried to learn it last time I went over this material, but I never succeeded. So I'll just call if LASSO. Okay, so that's this thing. And another form is simply to add this as a penalty and then sweep gamma here. And in this case it's called basis pursuit denoising or something like that, so. And I can explain why it's basis pursuit. If you are selecting columns of A, you think of that as sort of selecting a basis. So this, I guess – don't ask me why it's called basis pursuit, but that's another name for it. Okay. Let's do an example. It's actually – when you see these things, they are actually quite stunning.

And they are rightly making a lot of people interested in this topic now; although, as I said earlier, the ideas go way, way, way, way back. So here it is. I have a thousand long signal. And I am told that it only has 30 nonzeroes, so it's a spike signal. Now just for the record, I want to point out that a thousand choose 30 is a really big number. Okay. Just so the number of possible patterns of where the spikes occur is very, very large. For all practical purposes, it's infinitely large. And here is what's going to happen. We are going to be given 200 noisy measurements. And we'll just generate A randomly.

And there will be Gaussian noise here, okay. And then you are asked to guess X. Now, by the way, I should point something out. If someone walks up to you on the street and says, "I have a thousand numbers I want you to estimate. Here are 200 measurements." You should simply turn around and walk the other way very quickly, okay. Get to a lighted place or something like that as soon as you can, or a place with other people. The reason is it doesn't – this is totally ridiculous, right.

Everyone knows you need a thousand – if you are going to measure a thousand signals, you need a thousand measurements, okay – so at least a thousand, right, and better off is 2,000 or 3,000 or something like that to get some redundancy in your measurements, especially if there is noise in it. But the idea that someone would give you one-fifth the number of measurements as you have data to estimate and expect you to estimate it is kinda ridiculous, okay. Now, by the way, the flip side is this. If someone told you which 30 were nonzero, you move from five-to-one more parameters than measurements to the other way around.

If I tell you that there is 30 numbers I want you to estimate, and I give you 200 measurements, now you are 8-to-1 in the right direction, or you are 7-to-1 in the right direction. In other words, you got seven times more measurements than – everyone following this? Okay. So, all right, so what happens is if you simply give this L_1 thing – you just – well, you can see in this case you just recover it like – I guess it's perfect. I mean it's not completely perfect, some of these. I think the sparsity pattern is maybe perfect. It looks to me like it's perfect. Yeah, if it's not perfect, it's awfully close.

By the way, what that means is if you polish your noise will go down lower, and you'll get these very close. You won't get it exactly because you have noise. But the point is this is really quite impressive. If, in contrast, you would use an L_2 reconstruction, a [inaudible], and you would solve this problem, you can adjust gamma – basically, it never looks good, the reconstruction. But this would be an example of what you might reconstruct, something like that. So this is the rough idea, okay. So these are actually pretty cool methods.

I mean I don't really know any other really sort of effective method for doing something like this, for having – for saying, look, here is 200 measurements of a thousand things. Oh, and here is a hint, prior information. The thing you are looking for is sparse; please find it. And these work. I should also mention, unlike least squares. Least squares is kind of nice. It's a good way to blend a bunch of measurements and get a very good – it can work beautifully well if you have got like 50 times more measurements than variables to estimate. You use least squares. Almost like magic, it's a 263 level, right.

All of a sudden, from all of these crazy [inaudible] with noise, out comes like a head that you are imaging or something like that. I mean it's really quite spectacular. And it kind of fails gracefully. I should add something here. These don't fail gracefully. And I bet you are not surprised at that. So if you take this – which you can, all of the source is on the web, everything – and you just start cranking up sigma. You crank it up, and up, and up. And it will work quite well up until some pretty big sigma – you'll give sigma one more crank up and, boom, what will be reconstructed will be just completely – it will go from pretty good reconstructed to just nonsense very quickly.

So I just thought I'd mention that, so. Somehow it's not surprising, right. Okay. Let me mention some of these theoretical results. Obviously, I am going to say very little about it. They are extremely interesting. In fact, just the idea that you can say anything at all about it I find fascinating. But here it is. The problem is going to be that the set C is going to be very embarrassing. It's going to be an affine set. So basically, suppose you have Y equals AX, where the cardinality of X is less than K. And you want to reconstruct X here. Now, obviously, you need – the minimum you could possibly have would be K measurement.

So in other words, if someone comes up to you and says, "Wow, that's good. You have got my 30 non – my spike signal with 30 things. What if I give you 19 signals?" That's not even – that's not enough to get 30. So on the other hand, if the number of measurements is bigger than the size of the number of parameters, then we are back in 263 land, and everything is easy to do. So that's trivial. So the interesting part is where M lies between K, that's the sparsity – known sparsity signal and the number of parameters. And the question is: when would the L_1 heuristic that's minimizing norm X1 subject to AX equals Y.

I mean and notice how simple the set is, C if the set of X – AX equals Y. When would it be constructed exactly? That's the question. And actually, this – you can actually say things which are quite impressive. And it basically says this. It says that depending on the matrix A here – but there is a lot of matrices. Actually, there is a long – there is a lot of matrices that would actually work here. And there is actually all sorts of stuff known but what exactly what it is about the matrix that does the trick; it has to do with coherence or something like that.

And it says basically that if M is bigger than some factor times K – so this is sort of – if I gave you the hint, if I told you what the sparsity pattern is, you would need M bigger or equal to K. So the extent to which this number goes above one is how much more you need than the minimum if you had the secret information as to what the sparsity pattern was. And this is an absolute constant C times log N here. Then it says if this works, then the L_1 heuristic will actually reconstruct the exact X with overwhelming probability. What that means is that as you go above this, actually the probability of error goes down exponentially. Okay? That's it.

And some valid A's would be this. If the entries in the matrix were picked randomly, that would do the trick. If A was a rows of a DFT matrix, of a discreet 48 transform, then it would work. As long as the rows are not like bunched up or something like that, then it would work. And there are lots of others. So this is it. And these are beautiful things. I would take you about four seconds to find these on the web with Google, to find references and just get the papers. So, okay, so we will go on to the second part of this lecture, and that is going to be here. There we go. Great. Okay. So we are going to look at some more advanced topics here, so – and just other applications, and variations, and things like that.

So one is total variation reconstruction. I should add that this predates the current fad. The current fad, L_1 fad, it depends on the field. I mean statisticians have been doing it for a long time, like 12 years or 15 years now. People in geology, I think, have been doing it for 20. Others have been doing it probably 20 or something like that. But the recent thing, actually, was spurred by these results. And that's in the last five years, let's say. But total variation, this goes back, I believe, easily to the early '90s or something like that. So it works like this. You want to fit a corrupt – you have a corrupted signal, and you want to fit it with a piecewise constant signal with no more than K jumps.

Well, a simple way to do that is to trade off – X hat is going to be your estimate of your signal. You trade off your fit here – by the way, if this were Gaussian noise that would be something like a negative log likelihood function here. You would trade off your fit with the cardinality of DX where D is the first order difference matrix. And what you do is you would vary gamma. If you made gamma big enough, the solution X is constant. And then, of course, it's equal to the average value. As you crank gamma down from that – from that number, what will happen is the X hat will first have one jump, so it will be piecewise constant with one jump, then two, then three, then so on and so forth. Okay.

So DX1, by the way, is the sum of the differences of the absolute values of a signal – this is scalar signal for now. That's got a very old name. It goes back to the early 1800's. That's called a total variation of a signal – of a function – a signal, that's fine. And this is called total variation reconstruction. And there is a lot of things you can say about TVR reconstruction, but what happens is they actually are able to remove sort of high frequency noise without smoothing it out. We'll see how that works, like an L_2 regularization will just give you a low-pass filter; it will smooth everything out.

Now these, they are very famous here because they didn't – these were some of the methods do things like recover. These original from the wax recordings of Caruso or something, they actually reconstructed them. They got all sorts of jazz stuff from the '20s and reconstructed them using these methods. And they are amazing. I mean sort of the clicks and pops just go away. I mean they are just – they just – they are just removed. Okay. So here is an example. And so here we have a signal that looks like this. It's kind of slowly moving, but it's got these jumps as well. I mean this is just to make a visual point.

There is a signal, and the corrupted one looks like this. So there is a high frequency noise added here. Okay. And if we do total variation reconstruction, these are three values of gamma. And they are actually chosen – one is supposed to be like too much; one is too little; and one is not enough. But you can see something very cool. The jumps are preserved. So – and that's not like – that's not smoothed out at all. That's jump – that's smoothed out perfectly. Okay. And this is sort of too much because I have actually sort of flattened out the curvature that you saw here.

This is maybe you haven't removed enough of the signal, but you still get this jump here. And this might be just enough. By the way, if you were listening to this – in fact, I should probably produce some like little jpeg – I mean some little audio files or something like this so you can hear total variation denoising. It's very impressive. L_2 denoising, in a minute we'll see that, that's just low pass filtering. You have a lot of high-frequency stuff with everything just muffled. You hit a drum, everything is muffled. The L_1 [inaudible] will actually cut out this high-frequency noise.

But when someone hits a snare drum, it's right there. So it's pretty impressive. We should – it would be fun to do that, actually. Okay. Here is the L_2 reconstruction, just to give you a rough idea of what happens. In L_2 reconstruction, to really smooth out the noise, basically, you lose your big edge here. And this is sort of, maybe, the best you can do with L_2 or the best trade-off. And this would be – if you still – by the way, you still – in this case, it's not – that has not been preserved exactly. It's actually been smoothed a little bit. You just can't see it here. And you still are not – you are not getting enough noise attenuation, so just get a picture. Yeah.

Student:[Inaudible] that even this sounds very, very good.

Instructor (Stephen Boyd):This one?

Student: Yes, L_2 because that's the one we didn't do extraneous [inaudible].

Instructor (Stephen Boyd):Yes.

Student: This one sounds very good. And why should we go the extra half if we are going for L_1? I mean [inaudible].

Instructor (Stephen Boyd):Oh, in cases – I mean to do things like remove clicks and pops. And then, if you started listening carefully, you would find out this did not sound good at all. I mean not at all.

Student: Okay.

Instructor (Stephen Boyd):Yeah. Because you'd either hear this noise, right, or you start muffling this. And that makes a drum sound like – then you are not tapping a drum; you are tapping like a pillow or something like that. And it's no longer a drum. I mean just – so that's the – if you listen to these things, it's quite audible. We can adjust the parameters so the 263 methods work well, which of course naturally we do in 263. Wouldn't we? So, okay. So, okay. But actually, the total reconstruction variation is really done more often for images. And I believe it's also done even for – in 3-D. And I believe it's done even for 4-D, so for 3-D movies with space-time.

But we'll look at it in 2-D, and it's quite spectacular. So here is the idea. And this is going to be very crude. And I'll make some comments about how this works. So what it is, you have X and RN. These are the values on a N by N grid. So our R grid has about a thousand points on it. I mean it's small, but that's it. And the idea is I want – here is a prior knowledge is that X has relatively few – so X is sort of piecewise constant. In other words, it's like a big region. It looks like a cartoon. It's got big regions where it's constant and with a boundary. So everybody see what I'm saying? So that's the – it's cartoon looking.

It looks like a cartoon, or a line drawing, or whatever. All right, now this problem. You get 120 linear measurements; that's, of course, a big joke. You are whatever that is, six or seven times – I guess you are seven times under or six, whatever it is. You're some big factor under here, eight maybe that is, I don't know, eight. So you are eight times under sample. In other words, I want you to estimate 960 numbers; I'll give 120 measurements. These are exact. These are exact. So the way we'll do this is we'll say, look, among – this has, among all of the X's that are consistent with our measurements, that's a huge set.

In fact, what's the dimension of the set of X's that satisfy this? What do you think? What's the dimension on that? You don't have to get it exactly, just roughly. What 840 dimensions? Yeah. You got – you get 961 points. You get 120 measurements, null spaces on the order of the – is the difference, right. So, I don't know, you have eight. So this is a huge number of X's are consistent with our measurements; 840 dimensional set of images are consistent with our linear measurements. But among those, what we'll do is to pick one we'll do this. We would like to minimize – this is the sum of the cardinalities of the differences, and let me show you what that is over here.

And I'll explain in a minute how to make this better – look better, anyway. It's this that we have our grid. And basically, we would – we are going to charge for the number of times two edges – two values are different. And that's both this way and this way. So, for example, that big objective would be zero if the entire image were constant. Otherwise, everywhere where there is sort of a boundary, you are going to get charged, okay. So that's the picture. Now we can't solve that problem, but we can solve this variation on it.

Now, by the way, when you do L_1 this way on an image, and you just go – you charge for this way and this way, what happens is you are going to tend to get images – or you'll get things that will – they actually prefer like this direction or this direction, and you get weird things. I think we had that in a maybe a homework – no final – was it final exam problem 364? Was it? I can't remember. I think it was. We made Jacob's happy face. Midterm? Final? Midterm.

Student:[Inaudible].

Instructor (Stephen Boyd):What?

Student: Homework.

Instructor (Stephen Boyd):Homework. Okay, homework, sure. Anyway, so okay, so let's see how this works. So here is the – here is the total variation reconstruction. And the summary is it's perfect.

Student:[Inaudible].

Instructor (Stephen Boyd):Yeah. I know.

Student:I just [inaudible].

Instructor (Stephen Boyd):Great. Okay. Good. Good, okay. I forget, too. Wait until you see the [inaudible] 364c.

Student:[Inaudible] 364c?

Instructor (Stephen Boyd):Oh, yeah. Yeah. We are starting it this summer just for you. You are already enrolled. You'll like it. We're bringing the final exam back on that one, except that it's going to be every weekend, though, 24-hour. But you are learning a lot. You are going to learn a lot, though. Okay. So I – I mean this is – you get the idea. In this case, you recovered it exactly. So I mean these are kinda impressive when you see these things. Variations on this, by the way, are quite real. This is a fake toy example. You can go look at the source code yourself, which is probably like all of ten lines or something.

The plotting, needless to say, is many more lines than the actual code. These are actually quite real things. I mean there is stuff going on now where you do total variation reconstruction MRI from half of this – a third of the scan lines, and you get just as good an image. So, okay, and this is what happens if you do L_2. And I mean this is what you would imagine it to look like. That's kind of what you'd guess. And you can adjust your gamma and make the bump look higher, or less, or whatever. But it's never going to look that great. That's – this is your 263 method, so. That's essentially a least-normal problem. Yeah.

Actually, I'm sorry, there is no gamma in this problem. That's just least normal. Okay. So that finishes up some of these – oh, I should – I said I promised I was going to mention how these methods actually done. What you really want in image is you really want your estimate of – is to be approximately rotation and variant. So, in fact, you get a much better looking – visually now, if you really do this, not by just taking your differences this way, but you'll also take this difference here as well. And you will – and that thing, you'll divide by square root two or something like that.

And the other, the most sophisticated way, is to take your favorite multi-point approximation of the gradient and take the two norm of it and minimize the sum of the two norms of those gradients. That's the correct analog of total variation reconstruction in an image. And that will give you beautiful results that will be approximately rotation in variant. Okay. So let me talk about some other methods. This is also – this is just starting to become fashionable, the L_1. And there is other variations on it. And I think people call it beyond L_1 or something like this. And it goes like this.

So one way is to iterate, and I'll give a – we'll give a very simple interpretation of this in a minute. So I want to minimize the cardinality of over X and C. So what you do is this is instead of minimizing the L_1 norm, we'll minimize like a weighted L_1 norm. Now we have already seen good reasons to do that. The weights correspond – one over the weights correspond to your prior about the bounding box. So that's one way to do – one way to justify weights. But the idea here is this. You solve an L_1 problem, and then you update the weights. And the weight update is extremely interesting.

It works like this. If you run this L_1 thing, and one of these numbers comes out zero, this – then you get the biggest weight you can give, which is one over epsilon. What that means is thereafter, it's probably going to stay zero because it went to zero at first. Once you are zero, in the next iteration your weight goes way up. And then there is very strong encouragement to not become nonzero. What's cool about this is if X turns out to be small but not zero, so you are actually being charged for it cardinality-wise, what this does is it puts a big weight on it. And it basically makes that one look very attractive.

And it basically mops – cleans up small ones, just gets rid of them. On the other hand, if XI came out big, you are taking that as a heuristic to mean something like, well look, if you minimize an L_1 norm and something comes out – one of the entries comes out to be really big, it basically means, look, that thing is not going to be zero anyway. You are not going to drive it to zero. So therefore, relax, and basically says reduce the weight on it. So if it wants to get bigger, let it get bigger. So this is the picture. And this will typically give you modest improvement. Well, I mean it will actually give you real improvement over the basic L_1 heuristic.

And it typically converts in five or fewer steps. By the way, a more sophisticated version, actually, is not symmetric here with the weights. We'll see what the more sophisticated version is, but anyway. So here is the interpretation. So we'll work with a case where X is bigger than equal to zero. And we'll do that by splitting X into positive and negative parts where those are both non-negative. And the cardinality of X then, it's the same as this thing, I mean provided one of those is always zero. And we'll use the following approximation. Instead of – let me show you this. In fact, this kind of the idea behind all of these new methods that are beyond L_1.

I mean it goes back to – I mean all of these things go back to stuff that is very stupid. By the way, these things are very stupid, and yet it doesn't stop people from writing fantastically complicated papers about it, right, and making it look not stupid. But they are stupid. So let's go to one and stop there. Okay. So here is the function we want. It jumps up and goes like that. Here is our first approximation, not an impressively good – not what you call a good approximation. And so some of the – this one says you replace it with a log one plus X over epsilon. And, you know, if you allow me to scale it and change various things, that's a function that looks like this.

I'll shift it and scale it, if you don't mind. And it's a function that looks like that. Well, let me just make it go through there. There, okay, so it looks like that. And this little curve at the bottom, that's the epsilon like that. So it looks kinda like that. Okay? I drew it with epsilon exaggerated. I really shouldn't have. Let me redraw it, and it looks like this. And then you have got a little thing like that. That's sort of how you are supposed to see it. Okay? And you are supposed to say, well, yeah, sure, okay. This function here is a way better approximation of this thing than this, okay. What? You don't think it is?

Student: It’s not [inaudible].

Instructor (Stephen Boyd):It’s not convex. You are very well trained. Right. I can tell you a story. A student of Abass’s went into – they were just talking to Abass. And Abass said, “Yeah, but then you can do this problem and maximize the energy lifetime of this thing and blah, blah, blah, like that.” And the student stepped back. And he said, “Are you crazy?” And Abass said, “No. What’s wrong with that?” And the student looked at him and said, “That’s not convex.” Then he came and complained to me. He said, “What are you – what are you doing?” So you are right. That’s not convex. Okay. But it’s okay. Now you are okay. You can handle it.

So in fact, this method is – in fact, not only is it not convex, it’s concave. Now if you have to minimize a concave function over a convex set, when we did sequential convex programming, you saw that there is a very good way to do that. And it’s really dumb. I mean it’s – what you do is you take a certain X, you linearize this thing at that point, and you optimize. No trust region, nothing. And you just keep going. If you linearize this, basically you get this thing here. This is a constant, and it’s totally irrelevant when you linearize. And you actually get this. And in fact, part of that is a constant, too, like this part. Okay.

And in fact, it’s the same as minimizing XI over this. And guess what sequential convex programming applied to this non-convex problem, which is supposed to fit the cardinality function better, yields that exactly. Okay. So this is really an interactive heuristic. Sorry, it is. This is a convex-concave procedure for minimizing a non-convex function versus a smooth function, which is supposed to approximate the – what do you call it. It’s supposed to approximate this card function, okay. By the way, there is other – lots of other methods. And I can say what they are. Here is a very popular one. All of these work.

That’s my summary of them; lots of papers coming up on all of them. Here is another one. I don't know, here is one; you don’t like – let’s just do it on the positive. You don’t like X, how about X to the P for P less than one. So these functions start looking like that. And the small – if you make P really small, they look just like that card function. And by the way, this leads some people to refer to the cardinality as the L zero norm. Now let’s just back up a little bit there. And two of you have already said that, so you can say it. I cannot bring myself to say that because there is no such thing as an LP norm with P less than one because it’s not convex.

The unit balls look like this. And then I can’t even say it. You see? That’s what happens if you learn math when you are young. That is not the unit ball of anything. And you should not say that. You shouldn’t even – you shouldn’t say that. And yet, you will hear people talk about LP norm with P less than one. And it’s not convex. It’s not a norm, blah, blah, blah. So the methods work like this. In fact, you tell me. Let’s invent a method right now. How would you – how would you minimize this approximately, and heuristically, and so on? By the way, if you minimize that, you would get a very nice sparse solution, very nice.

How would you do it? You just linearize this thing. And what would you – and what, in fact, would you be doing at each step, and if you did the convex-concave procedure on this guy? You would be solving an iteratively re-weighted L_1 problem. Okay. And the only thing that would change is your weight update would be slightly different from this one. But your weight update would be reasonable. And I always do the same thing. What happens in a weight update is this. Entries that are big, you just say, ah screw it. That's probably not going to be zero, and you reduce the weight. Entries that are small, you crank the weight up.

If that thing is already zero, that's a strong inducement to pin it at zero. If it's small, thought, that's a – that makes that thing a very attractive target for being zeroed out. And that's what drives the cardinality down. Everybody got this? So that's the idea. Okay. It's a very typical example is you want to minimize the cardinality of X over some polyhedron. And the cardinality drops from 50 to 44, not that impressive. And if you run this heuristic, I guess six steps it converges, actually, after a couple, it will stop out at – no, sorry, let's see. Here we go. L_1 gives you 44. And the iteratively re-weighted L_1 heuristic gets you 36.

The global solution, probably, found for this problem in long time, later in the class, is 32. So just emphasize, again, we are not – we are not actually solving these problems. These are heuristics. But they are fast, and they are good, and so on and so forth. By the way, the fact that you are not solving the problem if the problem is – for example, is rising in a statistical context, I think means it doesn't matter at all. Right? Because it – you don't get a prize for getting the exact maximum likelihood estimate. Maximum likelihood estimate is just – is itself, in some way, it's just a procedure for making a really good guess as to what the parameter is.

And it's backed up by a hundred years of statistics. Okay. If you miss that – and by the way, even if you do perfect maximum likelihood, as any statistician or anybody who knows anything about it will tell you, you are not going to be getting the exact answer anyway. That's just some that – that's just some which asymptotically will do as well as any method could or something like that. That's its only – now by the way, for engineering design, that's a different story, right. You find a placement of some modules on a chip that takes 1.3 millimeters as opposed to – whereas, opposed to 1.6, that's real.

That's unlike the statistical interpretation. But still, okay. So let's look at an example of that. It's a fun example. It's just detecting changes in a time series model. So let's see how that works. We have a two term ARMA model, or I guess people call this – I'm sorry, it's not ARMA – it's AR – I think, actually, people call this AR(2). So it looks like this. Y of T plus two is equal to a coefficient times Y of T plus one, plus another coefficient times Y of T plus a noise, which is Gaussian. Now the assumption is this: these coefficients here are mostly constant. And then every now and then there is a change in the dynamics of the system.

And one or both of those numbers changes, okay. You'll be observing merely Y. And your job is to actually estimate A and B, and in particular, to find where the changes are.

So let’s see how that works. By the way, well, it doesn’t matter. I mean I was going to make up an application of it. And it basically says the changes could be – could tell you something about a failure in a system or something like that, or a shock in a financial system or economic system, something like that. Okay. So here is what we’ll do is we will – given Y, this is a negative log likelihood term here with some constants.

That’s a negative log likely – this is an implausibility term. Because, for example, if you run up a giant bill here, it’s asking you to believe that the V did some very, very unlikely things. That’s what this term is. And then here, we add in a – actually, it’s a total variation cost. It basically says it penalizes jumps in A and B in the coefficients. Now, by the way, if I make gamma big enough, A and B will be constant. Okay. If I make them zero, then I will make this thing zero because I can adjust my A and B, in fact many ways, to get absolutely perfect fit here. Okay. So here is an example.

Here are how A and B changed, so A is this, and then, I guess, I don't know, I can’t see now. But let’s say if T equals a hundred, it changes to a new value. B is here, and a 200 pops up here. So there is three changes. And you can sort of, visually, if you squint your eyes, you can change in the dynamics of the system here that, if you look left of there, you get one kind of dynamics. You can see a little – some – with a little squinting, you can see that the dynamics on the left in between a hundred and 200 looks different. And again, between 200 and 300, well, it helps that have I told you what happened.

But none of the – it’s certainly consistent. Now the interesting thing, though, is imagine I hadn’t told you this. I don't know that it’s that obvious. I mean certainly this doesn’t look very much like that. But would you really know that something happened here? I don't know. You could – I could have made the coefficients change in such a way that you would – your eyeball – you couldn’t do it. So if you run this total variation heuristic, on the left this is the estimate of the parameters. And I want to point out this thing is already like very good. It’s estimated the parameters to be here. It jumps down, and it does some weird thing here.

I can explain that a little bit here. It’s some little false positives. That’s a false positive where this thing jumps up. Every time this thing jumps up, there is a bunch of false positives in here, and some false positive jumps in here, and so on. But actually, you know what, this is not bad at all. Not bad at all. This is kinda – it’s kind of saying that there are weird changes in here and here. If you do the iterated heuristic, you can actually see visually exactly what happens. By the way, this guy is pulled down here because it’s charged a lot. It only makes this big shift for which it pays a lot in this objective to make the – to try to make this overall objective small.

But what happens is actually really, really cool. What happens is this. If you iterate it, this difference is really big. And on the next step, it’s going to get a less weight. So it basically says, oh, you really want to jump here? I am going to charge you less for the jump at this time step. Here, these little guys – by the way, where it’s flat, it says, okay, you don’t want to jump at all. I am going to make – I am going to charge you the maximum amount. That’s the one over epsilon in the weight. And these little ones, that’s what these L_1 – iterated L_1 heuristics do. They go up, and they clean up things. They just get totally nailed.

And that's the final estimate there. And you can see that it's much, much better. I mean you are actually tracking the parameters very nicely. You might ask why the error here? Why the error here? Why did it miss the time point here? And the answer would be because there was noise. That's why, but it's still – it's awfully good to do this. Okay. It works very, very well. Okay. And our last topic is going to be the extension of these ideas to make matrices and rank. So if you have – if you have cardinality of a vector – that's a number of nonzeroes, there is a very natural analog for matrices, and that's the rank.

And by the way, both of these things come up as measures of complexity of something. So in other words, sort of the complexity of a set of coefficients is something like – I mean this is very rough number of zeroes or something. And the complexity of a matrix also comes up a lot, and that's the rank. Now a convex rank problem, that's a convex problem except you have a rank constraint or rank objective, these come up all the time. And they are actually related. If you have a diagonal matrix and the rank of it is the cardinality of the diagonal, that's kind of obvious. But the interesting part is what's the analog of the L_1 heuristic?

And it turns out it's the nuclear norm, which is the dual of the spectral norm or maximum singular value. And it's the sum of the singular values of a matrix. So that's it. It's not simple, but that's what it is. It is the sum of the singular values. And that's the dual of a spectral norm, which you probably didn't know but it kinda make sense. Because somewhere in your mind you should have this map there that associates – well, it should associate LP and LQ, where one over P equals one over Q is one. But particular pairings should be burned into neurons directly. That's two and two, so dual of L_2 is L_2 type thing.

And dual of L_1 is L-infinity; dual L-infinity is L_1. These you should just know. So it shouldn't be surprising that if it's the maximum singular value, that's like and L-infinity on the singular value, roughly; that the dual norm should be the sum of the singular values. That's it. Now if a matrix is positive semi-definite, then symmetric – positive semi-definite, then the eigenvalues are the singular values, and the sum of the singular values are therefore some of the eigenvalues. That's a trace, okay; whereas, for a vector, if I have a non-negative vector, and I think the one norm it's the same as the sum.

So, oh, and by the way, that's why a lot of things would – I would still call them sort of L_1 heuristics, but you might sort of grip through the paper and never see L_1 mentioned because if a vector is non-negative, it's just a sum. But L_1 sounds fancier than to say L – you know, than the sum. The sum heuristic seems kinda dumb. Okay, so this is the – this is the nuclear norm. And we'll do an example. Actually, it's a very interesting example. It goes like this. You are given a positive semi-definite matrix. And what you would like to do is you want to find – you want to put this in a factor model.

Now a factor model looks like this. It’s an outer product, a low rank outer – a low rank part plus a diagonal. What this – I mean it would come up this way. It basically says that if covariance of a matrix, it basically said that that’s – that random variable is explained by a small number of factors. In fact, it’s the R – R is the dimension of F, the number of columns. It’s a small number of factors. And then D is sort of an extra variation. So this would be the factor model. Now by the way, there are some very easy ways to check factor models. If D is zero, how would you approximate – how do you approximate a positive semi-definite matrix as just low rank? How do you do that?

I give you a covariance matrix like the covariance matrix of 500 returns. And I want you to tell me – approximate it as a matrix of rank five, how do you do it?

Student: It’s the [inaudible].

Student: It’s [inaudible] symmetric.

Instructor (Stephen Boyd):So you should say – I can [inaudible] decomposition, but it’s the same as the SVD. So you take the eigenvalue decomposition, and you take the top five ones. So we know how to do factor modeling without this thing. But if I want to – let’s do one more. How about factor modeling plus – suppose all of the – suppose instead of D this was sigma squared I. Can you do factor modeling for that? How?

Student:[Inaudible] eigenvalues almost [inaudible].

Instructor (Stephen Boyd):Exactly. So the way you know it is you look at the eigenvalues of something, and you see – you would see five large ones and a whole bunch of small ones all clustered. And that would be your hint. Okay, so that would do that. That would be one way to do it. So you can do this with this. But when these actually numbers are all different, no one can solve that problem. Actually, it’s a hard problem in general. Well, in any case, this is the – this is the factor – simplest factor modeling problem. So C is a set of acceptable approximations to sigma. And they can be – I mean it could be simple, like some normal, or it can be very complicated and statistically motivated.

For example, it could be a Kullback-Leibler divergence, which would be this for a Gaussian random variable. Okay. And that’s just a very sophisticated way of saying that the two matrices are close. The two variances are – two co-variances are close. This is the one that would give you the statistical stamp of approval. The statistics stamp of approval would come from using something like that. Then a trace heuristic for minimizing rank is pretty simple. It goes like this: X plus D is your original matrix, and so your variables here are going to be X and – oh, oh, this should either be – well, I should either write – capital D is the diag of little D.

So it’s – but anyway, it doesn’t say that here which is weird. But that’s it. So this would be the problem. And that’s a convex problem. If you put rank here, it’s a convex rank problem. So we’ll look at an example. So here is an example where, in fact, I have a bunch of data, which are – well, we know it. It's actually generated by three factors, all right. So what happens is you get snapshots of 20 numbers. They are all varying. They are random. You look at these 20 – you look at a bunch of these things, they – you get a full covariance matrix, full rank. But it turns out that three factors describe it plus the diagonal elements.

By the way, in a – if these were asset returns, the diagonal elements would be called the firm specific. They would be – that's the firm specific variation. Basically, each – it said that you have a bunch of factors. I think one is the overall market, typically. And then you get some other things. Some very obvious things if you look at factor models in sort of finance you get these things. And then the D's are the firm-specific volatilities. Okay. We'll just use a simple norm – a norm approximation. And you get a trace – you get a trace heuristic that looks like it's a convex problem. And what we'll do is we'll generate 3,000 samples from here.

Now, by the way, we are estimating, I guess, it's 20 by 20 covariance matrix. You have got about 200 numbers. So you are maybe about 15 times as many numbers or samples are there numbers you are supposed to get. I guess each sample is 20. So, okay, it's a couple of hundred times in terms of the estimate, but you asked me in covariance, so okay. And this is sort of what happens. It's rank three. And what happens is, is you crank up beta. You start with a rank – by the way, the top rank is 20. But you immediately get a 15-rank model. Then, as you increase beta, that's the – that multiplies the trace thing or something like that.

What happens is the rank of that X goes down, and down, and down. You get a very steep drop down at three. And by the way, that's – this is the hint that rank three is going to be – give you a nice fit. If you keep going, if you increase beta enough, it goes from – it goes down there to two and then stays there. Actually, I guess this would go down to zero at some point. We didn't show beta [inaudible] large. What's interesting is as we scan beta, this shows the eignenvalues. And so up here you have 15 nonzero. And you can see at different values of beta, eigenvalues basically being extinguished. They go to zero.

And so what happens is, right here, you end up with three from here on. And in fact, these are the right ones. So if we take beta as some number in here like this, we actually do get a rank three model. You can do polishing in a case like this, too, obviously. Well, you can figure out what polishing is in this case. But in this case, if we take 0.1357, that's the tradeoff curve, you find that the angle between the subspace, which is the range of X and the range transposed is 6.8 degrees. And that we nailed the diagonal entries; we actually got the firm, specific volatilities within about 7 percent. So this is just an example of this kind of thing.

Okay, so this actually pretty much covers up this – the whole topic of sort of L_1 and cardinality. And the idea is instead of just thinking it as sort of a basic method where you just reflexively throw in an L_1 norm, actually these extensions show that, first of all, it comes up in other areas like rank, minimization. These internet methods, actually very few people know about them. They are not even used. Most people aren't even using polishing. Most people aren't even using the asymmetric L_1 stuff. So if you are interested in getting sparse solutions, there is actually better things than L_1 available.

And certainly, these things like these LP – I can't say it, LP norms – LP measures, I don't know. I don't know what word to say there that is okay. I'll just say it; LP, quote, norms, unquote, for P less than one. Those were these log approximations and these iterations. All of these things work. And I should also say – so I guess Emanuel Candiz and I had a long conversation like a year ago. And he said that L_1 is the least squares of the 21st Century. And, okay, it's a good – that's a good – I mean that's good, actually. It's not bad. I think it's a little bit of an exaggeration, but it's not too far off, right.

It basically says that they are going to be the same way everybody needed to know about least squares and throughout the 20th Century, eventually everybody did. And by the way, by least square I mean fancy methods like Kalman Filtering, and quadratic control, and things like that. If you call that least squares, then you know a lot of signal processing, and image processing, and all sorts of other stuff ended up being least squares, period. And I think a lot is going to end up being L_1 as people move forward. So, okay, so we'll quit here unless there is some questions about this material.

And then the next topic is actually going to be – we are going to jump to model predictive control. So we'll – that's going to be our next topic. Good. Good. We'll quit here.

[End of Audio]

Duration: 63 minutes

Instructor (Stephen Boyd):That's okay though. I see it puts a nice emphasis on, I mean, it's probably okay to say here's our problem, actually, it's an approximation. Here's the relaxation. Then when we run the relaxation, of course, on the real thing you get one thing, but then – actually, when you run it on the true system the relaxation thing does better than the other one. That's not uncommon.

Student: Yeah.

Instructor (Stephen Boyd):Perfectly okay.

Student: Yeah.

Student: It's Kreloff.

Instructor (Stephen Boyd):It's what?

Student: Kreloff.

Instructor (Stephen Boyd):Kreloff. I just spent five days with a bunch of Russians. I guess I'll have to remember that for next year. The left monitor's going to be right to you that I'm on. Does that mean I'm on? Something about this room, I don't know. At least I haven't said anything too horrible while being taped or whatever. Okay.

Today we're jumping around a different order of topics, but this is maybe, I think, the next topic that some people are working on, it's obviously too late for them for their projects, but we can at least cover the material and for people who are doing this at least it'll make a lot of sense. For other people, it's actually very, very good stuff to know about. It's widely, widely used. So it's called Model Predictive Control. In fact, I've been reading a lot about it the last couple of days. To sit through very long airplane flights, read a couple more books on it. It has got tons of different names, all different. Basically all the different areas doing this don't know about the others.

Often the notation and stuff is absolutely horrible. So, anyway, here's the basic idea. We'll get this in a minute. It's basically, I mean, there's a bigger area. Model Predictive Control is the name that comes from a relatively small community. In fact, I read one book and heard, I guess from some sort of member of this community, as referred to as the control community. That was as opposed to some others. Other names would be used. One that is a very good name in a sense is embedded optimization, real time optimization. Actually, the way that fits in the future, I think, is very, very interesting. A lot of people have their view of optimization locked into a model that's from 1969.

Either because they learned it in 1969 or they learned it from someone who learned it in 1969. They think of optimization as this big, complicated, very big thing that requires several Ph.D.'s, a bunch of super computers, and things like that. You really wouldn't use optimization except maybe to, I don't know, design a nuclear weapon, design the newest airplane, schedule United Airlines for tomorrow, or who knows what else. But these are the kinds of things you – or price derivative in the basement of Goldman Sachs. These are the things people think of when you think of optimization. You all ready know that a lot of these things actually are extremely reliable.

They're reliable enough to just work always, period, end of story. They're also very, very fast. By the time you say fast and reliable then it means you're automatically talking something candidate for imbedding in real-time systems, right? Because these, in fact, are the only two qualities you need to imbed in a real-time system. It has something that's gotta be fast, have bounded run time, and it has got to be extremely reliable. It can't require a Ph.D. to come over and say, "Oh, your algorithm's jammed. Let me change the number here. Let me restart it from a different point." You can do that if you're doing some big, big problem where it's important to run. That's fine. But if you're making a decision every 2.6 milliseconds you obviously can't do that.

The bigger name for this lecture would be imbedded or real-time optimization. Okay. So let's start. I think it's a very interesting area. I think you're gonna see a whole lot of it more in the future. Let's start with an optimal control problem and there's some very nice subtleties here and it's quite complicated. The idea, although, unfortunately, when it's taught many times the subtleties just aren't seen, but they're actually very important subtleties. So I'm gonna go over them a little bit because it actually takes some time to really understand what it means. So here's what we're gonna have. We're gonna have a linear dynamical system, so that's $x(t+1) = Ax(t) + Bu(t)$ and this u is gonna be our input or control or action.

We start from a state z. These input and controls have to be in some set U. The X's have to align some set script X and our objective is a sum over the entire trajectory of functions, this is called the stage cost or something like that, of this state and the control. Now, this is very traditional and you'd find this in like a vehicle control if you've taken a course on control or something like that, but, in fact, this describes tons and tons of problems, including supply chain management, things like scheduling and networking in networks and computer system. This just goes on and on and on. A lot of problems in finance look to be exactly like this. I mean, some don't, but many do. Even though I'm using the traditional symbols that you would find, for example, in the course in AeroAstro describing vehicle control, this can just as well be a finance or economic system or something else or a supply chain system.

So don't let the notation of the, as this book I was reading said, the control community fool you. This is much more general than controlling your local robot or something. Actually, it doesn't actually include your robot because the dynamics are linear. Okay. So the variables are an infinite number of inputs and we can also treat the states as variables too. In which case, we think of these as equality constraints. Now, the other option is to not think of the states as variables and to think of this as merely a recursion, which relates the X's, which are expressions, and, in fact, affine expressions to the variables, but these are actually equivalent. Okay.

Now, the data to the problem are this dynamic matrix A, this input matrix B, this stage cost function, and we'll assume that it's zero at zero. So, in other words, if you're at zero that's some costless point or something like that. We'll assume that these are convex and they include zero. You can actually change all of these assumptions pretty much and it just makes things more complicated. Okay. So the first thing one can have is something called greedy control and it's sort of the most obvious thing. It works like this.

What you do is you simply pick the input U to minimize the current stage cost here, that's what this does, over your allowed control actions and the only part of the future you intend to take into account is the following: Because of your action W, the next state is gonna be Ax(t) + Bw and the only accounting of the future you'll take care of is that this must be in this state constraint set, so that's Greedy Control. The basic idea is that it minimizes the current stage cost and it ignores the effect on the future. Well, except for this. It says that you can't do anything that would get you into serious trouble, namely violating the state constraint, on the next step.

As to whether or not you have violated the next step, that's entirely possible and too bad. Now, this method typically works very poorly. Of course, we can make it work very well, no problem. If A here, for example, has a very small spectral radius or very small norm it means that the X at the next state doesn't depend much on the current state and then that tells you that basically the dynamic system has what people would call something like strongly fading memory. That means that, in fact, Greedy is not too far from optimal because although your current action does have an affect it propagates into the future the affect goes down fast. For example, if the norm of A is point one that would certainly be the case here. The Greedy Control would probably work well. Okay.

If you haven't seen optimal control in dynamic programming – actually, how many people have seen this somewhere? Good, that's good. That's how many people should have seen it. I mean, this is sort of like 48 transforms. You need to see it maybe three times minimum, maybe four. Maybe it's more important actually, but I guess that's open for argument or something like that. You should see it three or four times. So this will be your, whatever, third or fourth time for some of you. If it's your first time that's fine too. Here's the way it works. You define V(z) as the optimal value of this problem right here. It is the optimal value of this problem. That's an infinite dimensional problem.

The variables here are the U's and the X's. It is infinite dimensional. It has an optimal value and it's a function of Z. That's an infinite dimensional one, but the optimal value of a convex optimization problem as a function of the right-hand sides of inequalities or equalities is convex in those parameters. The optimal value of this function is a convex function of Z. By the way, you know that rule very well. You know that rule in specific context, for example, the minimum of a quadratic function over some variables is another convex quadratic function. It's given by the sure compliment. So that's a very specific case where you can say much more about it.

In general, you know it's just true. Okay. So we formed this function V and that's called a value function. I should say that, there seems to be a weird – once again, MIT and other places split in typical notation. So some people use J as the value of the – I'll have to hunt down the source of this. J is the objective and then a lot of people use then V for the value function and at MIT it's reversed. They don't use V, this is J. So if you look in books from MIT or something the value function is called J. However, it is always called the value function or the Bellman function or, I forget what the Russian name for it is, but I think the Russian symbol is V, so anyway. Just a comment on notation.

This is convex. It's actually extremely easy just to show it directly. The important part is it satisfies Bellman or dynamic programming or Hamilton-Jacobi equation, many other names, Punctreargon equation. It says this, it's actually quite interesting. I'm not gonna go into a lot of the details, but it makes perfect sense and, in fact, it's extremely easy to show. It looks deep. A lot of people want you to think it's very complicated and deep and it's actually not. It basically says the following: It says that if you want to make the right choice right now from state Z here's what you do. Well, when you take an action Z you're gonna run up the bill l(z,w). That's immediate.

So the extremely greedy control, would say this: Choose W to minimize this. That would be the extremely greedy control. Well, that's not good because you have to at least take an action that will land you in an allowed state next step. That's the first plausible greedy method. As a result of choosing W, here's where you land: A(z) + B(w). So far we've all ready said that's feasible, that's okay. But the question is how should you take into account the affect of your current action on where you land? Well, in fact, if you imagine where you land, if you then simply carried out the optimal control from them it couldn't get any better than that.

It turns out this is what you should do. You should add to the current cost, the current bill you run up, plus the bill you would run up for landing in this state and thereafter doing exactly the right thing. So that's what this says. It says if you add these two together and minimize over W then this will actually equal V. It's the optimal thing and not only that, but the argmin of this thing is the optimal U, right? It looks complicated, it's really not. If you take an infimum, you can minimize in any order and it doesn't make any difference, right? So that's really what it is.

So this is the dynamic programming equation. By itself it says absolutely nothing, doesn't help in the slightest because then you'd say so what's V? What is interesting is that in some cases you can actually find V. Once you've found V you actually have the solution here. And there's one thing I want to point out about this, this optimal input is, in fact, a function of the state x(t) because although it's complicated that's a function of the current state. So that's what it is. In other words, that the optimal input has to form some function phi of x(t). State feedback form would be the common way to say that. That's quite interesting.

And here, as I said, the interpretation is this: That this term V(Ax(t) + Bw) is very important. It's basically the accounting term needed to completely accurately account for the effect on the future cost of your current action, so that's exactly what this thing is here. Okay. So one famous case where this actually leads to a numerical solution, but basically a solution, is this: You have linear quadratic optimal control, so the stage cost is X transpose QX plus u transpose RU. It's easy to add a cross term between X and U and it's easy to add a linear term and all that kind of stuff and this is just like exercises. Q is traditionally positive, semi definite and R is positive, definite. That's just to keep the formulas simple. Okay.

Now, in this case you can actually solve the problem by dynamic programming. It turns out the value function's quadratic. That's hardly a surprise, right? Because you basically have an infinite dimensional quadratic problem, an infinite number of variables. You know that for a finite dimensional quadratic function if you minimize over some variables the result is quadratic in the others. In fact, the result is a sure compliment in the others. You know that all ready. This merely extends that to the infinite dimensional case and, in fact, sure compliment is the right thing because it's very, very close. In fact, if you stare at this long enough you'll see a sure compliment here. I should have written it that way. I don't know why I didn't. You could take the A transpose and the A out of these two terms and you will see, I don't know how good your sure compliment visualization skills are, but you should see a sure compliment.

Anyway, it's plausible, right? Because a sure compliment is blah minus something, something inverse, something transpose where the third something was the same as the first something or something like that. Anyway, you know what I mean. So, anyway, that's a sure compliment, so it shouldn't be surprising. The value function is quadratic in this case and, in a sense, it is a sure compliment of an infinitely big positive quadratic function and you minimize over infinitely many variables leaving just N left and the result is, in fact, a giant infinite dimensional sure compliment, but as an explicit solution. This is the Bellman equation now written out. Because they're quadratic forms it just says the two matrices are the same and you get this.

Now, by the way, there's no reason to believe that you can solve this. It's a quadratic matrix equation, so there's lots of ways to solve this. One just by iterating and things like that, so I'm not gonna get into that, but this equation can be solved quite effectively. In this case, the optimal policy is a linear state feedback. It just looks like this. U* +k(x(t)) where k is this thing here. You can actually see all sorts of various interesting things here about what happens when P gets big and when it's small and all that kind of stuff. You can actually work out a lot of the behavior. This thing, I think a bunch of people have – actually, how many people have seen this? I'm just sort of curious. Okay.

So, if you've seen this you probably know this is extremely widely used. Even in cases where systems aren't linear and so on like that and it is unbelievably effective. I mean, state feedback gains like this just work really well. In particular, it's sort of like least squares. I mean, it's almost like magic in the sense that when you synthesize state feedback gains like this – I mean, this is not interesting if this data is small and U is small, but if you've got 20 dimensions and five inputs that's 100 numbers in there and 100 numbers just come out just like calculating a pseudoinverse and then doing image processing. It would take you a long time to adjust those hundred numbers by hand to get something that even works let alone works so well. This is widely used. Okay.

Now we're gonna look at this finite horizon approximation and this is quite straightforward. What we do is instead of solving this infinite dimensional problem what we're gonna do is we're simply gonna run it out to time capital T, that's some horizon. What we'll do is we'll insist that when you arrive at the end of this horizon you drive the state to zero. Now, once the state is zero you have the option of putting in input zero, zero, zero after that. We're essentially looking at a finite dimensional subspace of possible inputs. The inputs go out to a certain amount of time, they drive the state exactly to zero, that's this. Thereafter you apply zero to the whole thing.

So this would give you a valid input on infinite horizon. That's what this is. Okay. Now, then you have a finite dimensional convex problem. We can solve that easily and this will give you a sub-optimal input for the original optimal control problem. Why? Because it's basically you're looking over a finite dimensional subspace. It's the subspace of signals of inputs and states that banish after time T. That's a finite dimensional subspace. You solve the problem over subspace you certainly get an approximation. You get an upper bound. Let's just do an example of a system with like three states and two inputs. It really doesn't matter.

It's just completely random. Just quadratic stage cost and we'll have a limit on the state. None of the components can be bigger than one. Same for the inputs, they have to be less than point five. Who knows why. And then here's our initial input and the optimal cost turns out to be 8.83. If you're curious, I can tell you why that's not an approximation. That's the exact number. I'll explain it in a minute. Not that it matters, but let me get back to that in a minute. So here's the cost versus horizon. In this problem it turns out it takes ten steps. Notice if you say get to the origin it's infeasible for ten steps because you have some state, it propagates forward in time, you have an input, which is limited between plus/minus point five. There's this limited amount you can do.

By the way, those of you who are in 263 I should say that I made fun of it then, but I'll make fun of it even more now. This very traditional idea of like controllability and all of this. I don't know if you remember this. If you've blocked it out that was the right thing to do because, I mean, it's perfectly okay. It's the first thing you ask. Can you wiggle the input to take the state to zero or something and you get these kind of silly things involving ranges and all that and that's all fine, but the real question is not can you maneuver the state to zero in capital T steps? The real question is, can you do it with an input signal that's less than minus point five? That's actually a real question. That one's of interest and no stupid theory of controllability is gonna tell you that. Okay?

In my opinion, this is the real theory of control. This is the first theory that's actually useful. This is with b, a, b, a squared and b and all that. The Kreloff subspace, as we now know it. Being full rank and everything, I mean, I don't know. It's classical and, as far as I know, it doesn't really have any actual use, but okay. So here's a real controllability analysis. It tells you can you hit the origin. It says that for even in nine steps you can't do it. It's infeasible, so the cost is plus infinity. In ten steps you can do it at a cost of 11.5. Then in, I don't know what this is, in 11 you can do it down this way. You see something here that this is converging very, very quickly. This is very common.

It’s not universal. I should say, obviously this is monotone decreasing, but, in fact, there’s a lot that would tell you that this thing is gonna converge very quickly. In other words, it makes sense because the optimal inputs, I’ll show you what they look like. Here’s the actual infinite dimensional optimal input and you can see very carefully what happens here. The input looks like this, but then after a while it’s dying down like this and then that’s dying down like that and, of course, this is falling off exponentially. Here it’s not gonna take – if I tell you, in fact, you have to be zero or T=30 it takes very little perturbation of this input signal to drive this state to zero here and not have it be merely small. Takes just minor perturbation.

That’s why these things converge very, very quickly. Okay. So this is the cost versus horizon. Actually, people who have had these courses, I don’t know, 207, 210, anyone know how we calculated that line? This is just for those who have the background. I mean, look it would be perfectly okay to just do this, actually. Perfectly normal would be to solve this for T=50 or 100 and numerically say that’s it. That’s perfectly acceptable. But, in fact, you can calculate it exactly. I mean, up to this numerical round off here. Do you know how we did it? Do you know?

Student:[Inaudible].

Instructor (Stephen Boyd):This is with constraint. Yeah. So here’s how you do it and this is just an aside, but just for fun if you have the background. You can show in a problem like this that you can compute an ellipsoid in state space such that if you’re in the ellipsoid the optimal control is, in fact, coincides with the linear one because you’ll never saturate, never in the future. Therefore from then on it’s given by an explicit formula. In fact, it’s this formula right here. All right. This is an advanced topic, so just ignore me if you don’t know what I’m talking about. Also, I’m going the wrong way here. There we go. So it’s given by this formula and once you have that you can actually evaluate the cost to go by solving the Lyapunov equation. Okay.

If you knew what I was talking about, fine. If you don’t, just ignore what I just said. Okay. And if you look here you can see all sorts of things. This is the smallest amount of time you can possibly drive the state to zero and you can see this is really quite different. The input is quite different although, actually, curiously it doesn’t start off that differently here, right? I mean, it basically says I’m just showing you U, but I think maybe there’s several U’s. I forget and I don’t know why we’re showing two inputs. Okay. So this is one of the U’s. I don’t know which it is. We’ll fix this typo. It’s either U1 or U2. I mean, not that it matters, right?

You see it does something and then it pulls back. This is all because you have to drive the state to zero. There are two other states as well that get here from the zero up here at ten. Okay. All right. Now this brings us to Model Predictive Control. This is something everybody should know about this. Unfortunately, it’s considered part of control. It’s not even mainstream control because it’s considered part of industrial control. I think it’s really stupid. It’s incredibly effective for all sorts of things. It goes like this. Here’s the way it’s gonna go. There are some subtleties, so I’m gonna take some time to do it right.

It’s at time T and here’s what you do. You solve the following problem. You have a planning horizon capital T. From time T forward, capital T periods you are going to assess a cost, you will – all of this is essentially a planning exercise and it’s gonna work like this. You’re gonna propagate forward the dynamics of the system. You are gonna start at the current state. So you’re in a current state at X=T. You will do a planning exercise, which will actually propagate the state forward, capital T samples. You will insist that capital T samples in the future you must hit zero. By the way, there are variations on this so-called terminal constraint, but the easiest is just to put that there.

You’ll do this planning exercise. Literally, at time T you will work out an optimal control that would take the state to zero in capital T steps starting from there and so on. By the way, at this point the U’s here they constitute a plan for the future. If you were to execute that plan, then the state would indeed be at capital T plus capital T it would go to zero. The clever part is this, you’re gonna do a complete planning exercise and you’re simply gonna use the first input. Then you’re gonna step T forward one unit and you’re gonna re-plan, again. Your horizon has just moved and that’s how this works. It’s always like capital T is 30. You’re always planning to drive the state to zero in 30 steps at all times. That’s the way it’s working.

By the way, another name for this is receiving horizon control, finite look ahead control, dynamic linear programming because in a lot of supply chain problems these are linear or piecewise linear, so it’s got lots and lots of names, including it probably comes up in lots of places where it doesn’t have a name because it’s kind of so obvious. Okay. So that’s the idea. Now, the fact is this is, actually, also a state feedback. It’s a fantastically complicated state feedback. Well, it’s not that complicated. To tell you the truth it’s piecewise linear. We’ll get to that in a minute. It is quite complicated because it basically takes a bunch of data, namely x(t) and solves a big QP in order to calculate a full trajectory.

So I’m just trying to think if I should draw a picture. Well, why not. I mean, it’s the basic idea. I guess it’s kind of obvious, but just to make sure. The way it works is here’s time and you’re right here. What you do is you make yourself a plan of input, something like that, that will take a state to zero at this time. So this is the plan that you execute at this time step. You then simply execute this. That’s the U that you actually apply. You then step forward one to this point, but now your horizon has extended. There are other versions where it doesn’t. Your horizon has extended here and now there’s no reason to believe that you’ll do this. If you did this, you would actually take it to zero, but now, actually, what’s happened is your requirements have relaxed. You had 30 steps to take the state to zero before.

You step forward now and now instead of 79 it’s 80 you have to hit and, in fact, this one might relax. It might be I won’t be so aggressive. It’ll be slightly lower, so you can actually watch this, and then it would take it to here. Okay. So this is the basic idea. It’s complicated. I mean, it’s really interesting. In this case, you would solve a convex problem every step. For that baby example we have, here’s the way it works. Well, this is the MPC compared to the optimal. You see something that’s totally insane here, which is basically if you tell this thing plan ten steps in the future – by the way, if you ask it to plan nine steps in the future it can't even hit. It's not even feasible.

So you'd say plan at all times I would like you to act as if I want you to file a full, legitimate plan that ten steps in the future would take the state to zero. Here's how well it works compared to optimal. But if you say like 11 or 12 it's basically optimal. Yeah?

Student: So the way the graph is drawn it seems a little bit unfair to put them both on the same horizontal scale because one is [inaudible] than the other isn't it?

Instructor (Stephen Boyd):Yes. Right, right. Yeah. We do have to explain a few things. This one was calculated, as I said, by solving a QP big enough and then adding on an N from a Lyapunov equation. That's correct. And this one here, each point on that curve was done by solving one QP. Very good point. This thing was done by solving a QP for whatever 40 steps or 50 steps or something. You are right. The computation involved here is much more. That's correct. Yeah. Actually, even though it's a baby example and it's not always this dramatic there's a hint here that things like this work really, really well. I mean really well. The reason is actually pretty simple.

It does make sense. It goes something like this. As you pointed out, you're solving 50 QP's here basically. You're solving 50 different quadratic programs or something like that. All the time the QP you are solving is that you're making a plan of action for T steps in the future. Let's make it 12, so you're like out here. Basically, you're doing flawlessly. Basically, at all times, you're filing a plan of action to take the state to zero in 12 steps. What happens next? You then execute only the first input. You step forward and now instead of saying, well, it has to be zero at that time, that time when it has to be zero has shifted. Okay?

Let me explain why this works well. There is a sense in which at each step you basically compute a whole action. Basically it's a whole action plan is what it is. It means that if you were to get the signal all over, done, quit, stop, everything like that, then you would execute that plan in 12 steps or whatever it is. Capital T steps later you'd bring the thing to zero. So at all times you have sort of a viable plan that would halt the system in 15 steps. That's kind of the – or whatever capital T is, right? So, in fact, we have T=15. You're calculating 15 values of the input, but only one you use. Only the first one. The other 14 values of the input is nothing but a planning exercise and its only reason for being there is so that you don't do something stupid now that messes you up for the future.

That is the only reason for that planning to be there. By the way, once you realize that that's the basic idea, that the reason you're making a full planning is just so you don't do something now stupid with respect to the future, then it starts making sense that you could do a crappy job of planning and you would expect it to actually work. I mean, this sort of makes sense. Imagine this in some kind of a cash flow problem, right? Where you're managing some cash flow and that'll actually be something for next lecture, which is Stochastic MPC, but in a cash flow problem you would have inputs and now I've got expenses, I've got income, and stuff like that. What you do is you simply do some planning now. What you do now has a huge effect on the future, right?

You invest in some fixed income security that matures in three months it's gonna be a cost now, but it'll be a benefit in three months. You run your cash balance down real low now to pay for something, then you're gonna have to borrow and that'll be expensive when you have expenses come up. Everybody see how – I mean, it's the exact same problem. So MPC there would do just what you would imagine. You'd basically do, in fact, what you're required to do if you're a CFO or something like that. You, actually, have just a plan. You make a plan for the next 12 quarters on cash flow.

Now, so far everything's deterministic, so we, sort of, assume nothing's gonna change. We'll see that the real advantage comes when, in fact, things are not what you predict them to be. That's Stochastic control. That's the next topic.

Student: Is there a way to solve this problem with [inaudible]?

Instructor (Stephen Boyd):Uh, if they are changing, but linear, yes. That's no problem. But if they are non-linear it's not a convex problem and you know what my recommendation would be then? I don't know. Lecture whatever it was, sequential convex programming. It'll work quite well. Yeah?

Student: If the dynamics are changing, do you have to keep solving more feasibility problems to figure out your horizon? It seems like before you'd do this problem you have to figure out the feasibility problem to get your minimum –

Instructor (Stephen Boyd):Right.

Student: – or you can overshoot it, right?

Instructor (Stephen Boyd):Right. We'll get to that. Generally speaking, if you're running into feasibility issues your horizon is too short. The other thing is that when you run this, believe it or not, it'll work perfectly well if you violate feasibility. So if instead of saying x(t) + T is equal to zero, I put a big charge on capital X(t) like ten times its absolute value or one norm or something like that. This would work well now even, believe it or not, or tolerably it would work out here because it would just try its best to get that state zero. I mean, it wouldn't get great performance, but it wouldn't just like stop, right? So, yes. When you do these methods, you have to deal with gracefully, right?

Because if you do a plan for the future and someone says let me see your plan and there it is. It's like nan, nan, nan, nan, nan, right? That's, yeah. So you don't – you have to relax all the things. This just makes it easiest to present.

Student: Just a question. Has anybody looked at like different [inaudible] forms so you could reuse some of the computations you had done?

Instructor (Stephen Boyd):Excellent question. Okay. When this is running there should be something that's very clear. That at each step, if you're the CFO, you worked out basically how to wind up operations in two and a half years, right? But do so, by the way, minimizing your total expanse or maximizing your lifetime or your time to live or whatever it was. I mean, it's a very simple thing. Now you step forward and you're actually solving almost the same problem, right? That's your point. That can be exploited unbelievably well by using a warm start method. Then that gets back to your question, which is the computational complexity.

It is true that you're solving 50 QP's here, but you're solving 50 QP's and each is only a very minor perturbation on the other and, in fact, if you use a warm start method I guarantee it'll be two Newton steps to go. Basically, what happens is if someone says, "Okay. Now T is 17. Let's figure out what to do." And you say, "Okay. Well, what I have to do is I have to do a plan from T=17 to T=29." That's if capital T is 12. Okay? But you just did a plan from 16 to 28. So you take that plan and that's your initial point for your new plan. Two Newton steps I promise you, three, I don't know. It's all over, done. So, in fact, these things are fast. Okay? We'll get to that.

Student:[Inaudible] about the [inaudible]?

Instructor (Stephen Boyd):Yeah.

Student: This is very deceiving because what you're saying when the finite horizon control problem relate in 15 steps you are going –

Instructor (Stephen Boyd):Oh, you mean compare this one to this one? Yeah. Right.

Student: Another one is a strategy to compute, like, a somewhat different strategy to compute the optimal –

Instructor (Stephen Boyd):I agree.

Student: And you are actually going to reach near zero or some region near zero within terms of the radius near zero in some, like, 100 steps or 50 steps, not really in like 10 steps –

Instructor (Stephen Boyd):Right. No, I, no, but technically – okay, that is true. But let me put something out. Technically all three of these are showing the same thing. This one doesn't depend on horizon because it's the exact global optimal.

Student: Right.

Instructor (Stephen Boyd):Okay? Now, this one says bring that state to zero in 15 steps and then stop.

Student: Right.

Instructor (Stephen Boyd):That's a viable U. I mean, it's a U, it brings a state to zero, and it just sits there. It runs up a bill.

Student: But if you look at –

Instructor (Stephen Boyd):It runs up this bill. What's that?

Student: If you look at your first MPC and you actually look you will see that as if you are incurring the cost of 11.5.

Instructor (Stephen Boyd):Right.

Student: And then it would suddenly realize that I only used the most important then my cost is going to use –

Instructor (Stephen Boyd):Right. Well, what happened was at this point what happened for this MPC thing, on the first step, it's actually quite rushed, right?

Student: Yeah.

Instructor (Stephen Boyd):Because it's basically saying you minimize this cost, take this state to zero in 11 steps. It basically in nine steps is impossible. It's not even feasible to do, so it's feeling quite rushed and it kind of has got some input that just barely does it. Then what happens is, in the next step the horizon moves another step forward, right?

Student: Right.

Instructor (Stephen Boyd):So I think they're perfectly okay. I don't know. I think I'm not – they're showing different things, but it's okay. Here are the trajectories generated. This is where MPC with T=10 and, I guess, the point is here that at least visually, I mean, you know all ready you've seen the plot, they're kind of the same.

Student: Do you build anything like games like this where you have two players –

Instructor (Stephen Boyd):Yes.

Student: – and both of them –

Instructor (Stephen Boyd):Yeah. Yeah. So people have done games too. That's right. Yeah. Yeah, infinite versions of this and all sorts of other stuff. Okay. Here if you look carefully, I mean, you can see that these are – well, they have to be close, right? Because the costs were close, but, I mean, they're shockingly close. But the point is that this is calculated in a very different way. This is calculated by solving one QP essentially infinite horizon, let's leave it that way. Here. That's what this does. This says that this point was found by saying wind up in ten steps. Then it moves forward. It creates a whole plan for winding up in ten steps, moves forward in one step, now it's T=2 and it says now given the state you're in plan to wind up in ten steps.

So what's shocking is that each planning exercise is stressful in the sense that it's not even feasible for nine steps to get there. Well, it might actually start being feasible once the state has been reduced, but at first it's not even feasible. Each planning exercise is stressful, but the resulting input generated by sequence of stressful planning exercises is actually superb. I mean, this obviously doesn't always happen, but this is pointing out something which is correct in practice and that is that these things work like spookily well. I mean, if you have to solve things like this you should do this. Okay.

Now, I've all ready said some of this. It's got lots of names in the 70's. This was propagated all across chemical process plants and it had the name dynamic matrix control. You might want to ask why. The matrix means that they wrote a big matrix that related the future inputs to the future outputs and I'm not kidding. That's what it is, so that was heavy technology, but widely used. There was like four companies. They're used just everywhere now. So you go to a big chemical processing plant it's almost certain this is what they're using. Let's see. And they can get quite fancy. By the way, I remember talking to someone from Shell Canada and he said that they actually have in these things that are – so I'm talking about big giant process plants, right? By the way, this is very impressive because these are also very dangerous, right? You send the wrong signals to a giant cracking tower and you can be very, very sorry.

Also, they're extremely high value assets as you can imagine, right? So he said that, actually, in the QP, there's coefficients that go into that QP into the cost function come from analysts and things like spot futures prices and things like that for the future. Then I pointed out to him, I said that means that some guy in Chicago who raises a card, it's not done that way, but, anyway, raises a card on something, actually, more or less directly ends up affecting valves and pumps in Ottawa. He thought about it for a minute and he said yes, that's true. So I think it's kind of cool actually. I think it's cool anyway. Okay.

It has lots of names, rolling horizon planning. A lot of people do it and don't even say it so that's the other thing. Now, actually, what's interesting is that it's propagated into industries where the dynamics are slow. So, in fact, I just recently found out it's what's used, although not called MPC. It's used in so-called revenue management. Do you know what that is? That's basically these sleazy ways companies get to part with most of your money possible and you'll be happy to know this is what's doing it. This is how they release the number of airline seats at this price and then they change it and then more and then more and then they sense desperation like you say, "No, I need to be at this meeting in Chicago." They go like, "Oh, it's really expensive. I'm sorry about that." And you sit next to someone who paid 1/12 the price. The euphemism for that is revenue management. They're managing it all right. At least according to my friend at business school, these are the methods used. Although no one doing it would recognize the terms Model Predictive Control. They'd say never heard of it. They would call it revenue management and it would be this. Okay. All right. Let me say a few more things about it. Actually, there is something very important about this that I should say and I think it's extremely relevant. I think you hit it right. You went right to the key point. The key point is in these systems in these methods if you're running control you're solving a convex problem at every step.

Now, if you think about control and things like that, things that run in real-time, that is very non-traditional, right? So the way controls that were implemented in many industries up until recently, and included in many now, is the control which runs at real-time is just a very small amount of calculation, right? You spend a billion dollars on all sorts of stuff to figure out some control gains, but then that actual control code is like 14 lines of C. Got a couple of switches in it, couple of flops, and things like that and then it can be checked for correctness like a zillion times or whatever. Then it's installed in a flight controller. Not, by the way, because it's updated so fast. These things are only updated in like 50 hertz or something like that, so you have 20 milliseconds. It's just for tradition and various other things. Everybody see what I'm saying?

So the most common type of control is like PID, which is Proportional Integral Derivative. Basically requires about three multiplies per step or something, right? You get the new output compared to your target. That's your error. You do have to add that to the previous integral error, right? And you multiply by three numbers and that's what you do. Everybody see what I'm saying? When computers were slow this all made a lot of sense. People who did traditional control ignored this entirely through the 70's and 80's because they said it's a special thing. I mean, it works because chemical process plants have very slow dynamics that go for like 24 hours and if you have 15 minutes you can go ahead, solve a QP every 15 minutes. What do we care?

If you're doing revenue management solve a QP every 15 minutes. What do we care? LP in that case, right? Go right ahead. So people who did sort of the real-time control said, "Oh, no, I work on robots. I work on jet engines. I do networks. Our decisions have to be made at the millisecond or microsecond level. I have to schedule processes. I don't have 15 minutes to figure out what to schedule next. I've got to make decisions every millisecond." So there's a whole bunch of areas like that where whether those people know it or not this is coming for them. Mostly they don't know it, but this is coming. Okay.

So lets look at some variations. I've all ready mentioned this that instead of insisting that the final state be zero you can make a final state cost – make up some V head equals V. By the way, if the final state cost were exactly the cost to go function then, no matter how short the horizon, you would actually get the exact optimal input. In other words, even if it was one step – well, that's literally Bellman equation. The Bellman equation says let's do some planning, ready? Let's plan only what you're gonna do right now, but then you say, well, how do I count for the future cost? And you go by the exact effect on the future cost that would be V. If you put that accounting in and optimize Greedy locally in time you get the exact optimal. Everybody see what I'm saying?

So you might imagine, for example, the end point condition x(T)+T is zero. That is effectively a V had. It's the following V had. It's zero when X is zero. That's plus infinity otherwise. Not exactly what you'd call a superb estimate of the cost to go. Anyway, if you can put in an approximation of a cost to go here it'll work really well. For example, if a problem is like linear quadratic and zero is in the interior of the thing, you can actually solve the Ricatti equation and put X transpose PX as this thing and you'll just get amazingly good controls at that point. So that's just one thing. It also deals with the non-feasibility issue quite gracefully.

In other words, that way when someone says can I see your plan it's not gonna be nan, nan, nan, nan, nan as inputs right is. Okay. Again, on that topic, you convert hard constraints to violation penalties because you don't want to just say I can't make the plan. What you do is you just violate some penalties. What's really weird about this is you'd think things would work poorly. You can actually run MPC where all of your planning exercises are, actually, infeasible. If you freeze it and you say, "What are you doing?" You say, "I'm planning my actions for the next 14 steps." You go, "Great, what are you gonna do?" "Well, I wanted to make the final state zero, but it's infeasible. Also, I'm violating a few things. I'm violating the dynamics equations and things like that." And you're like, "You call that a plan?" And you go, "Hey, it's real-time."

Shockingly those things actually work really well. It kind of makes sense because if you think of it this way, if you think what's the plan for? The plan is there only so that you don't do something stupid now with respect to the future. So if you're planning is not perfect – well, I guess if it's infeasible it's a little bit worse than not perfect. I mean, at least it's some kind of rough plan. You couldn't execute it, but anyway. All right. Another variation is this. You can solve the MPC problem every capital K steps. Basically what happens is that at T you plan out capital T steps and you plan out a whole trajectory and you execute the first five. You just do those. Then after five steps you see what state you're in, there's no disturbances here so the truth is you know what state you're in.

Your horizon also shoots back five steps and you re-plan, so that's another variation on these. There are lots of variations. Okay. Now, let me talk about explicit MPC. This is a special case for these. The loss function is quadratic and then these are polyhedral. The state and input constraint sets are polyhedral. Then it turns out that phi MPC is actually piecewise affine. Okay. It looks shocking or cool or something, but it's not. It turns out that the solution of any quadratic program is piecewise affine in the right-hand sides of the constraints. That comes straight from the kickety conditions.

By the way, this comes up in machine learning also that when you do an L1 regularized problem and you put your lambda there, you can say lots of things about the X, like that it's piecewise affine in lambda and you can say various other things about it, but it's a good thing to know and it's not hard to see. Theoretically, it says that if you're gonna solve a QP with many instances of the right-hand side you can actually pre-compute an explicit lookup table for the solution. Now, you can do this for any QP for any application. Now, what happens then is you simply calculate these regions and then your code looks like this. Lets call your parameter P.

P comes in and the first thing you do is you find out which region P is in and then you simply apply KP + G where the K and G are the correct ones for that region. Okay? So that's how that works. This is practical for very small problems only. I mean, certainly an empty C. I mean, these are things like two and three and things like that. Small number of constraints. Now, people have pioneered in the last, like, five or six years methods that actually analytically construct these things and they make very pretty pictures. You can type explicit MPC into Google or something and you'll see beautiful pictures with a two-dimensional states base and some kind of tessellation and 180 regions, and they actually use these things.

They've used them in anesthesia. They've used them also in switching power supply. Obviously, these things can get slow very quickly, which is kind of the – the whole point was to make it fast or something. Again, this comes from 1969 mentality. 1969 mentality was solving a QP is a big deal. That's 1969 mentality. Big thing. You need a big super computer, you need expensive software, a couple of Ph.D.'s to sit around, big 208 volt three phase, 50 amps. People just think of it like it's a big deal and, as you all know, it is nothing. You can write one in Matlab in 20 minutes. You could actually. It's just not that big a deal.

You can write one in C in half an hour or something like that. So a lot of it is sort of mismotivated by that. Now, you can do things to simply these things and get quite a good control. There is another advantage to an explicit form and the explicit form is actually quite traditional in control. That would be called gain scheduling or something like that. The person who is traditionally trained to make sure that before some control code is downloaded into something dangerous or expensive they'll be very familiar with a switch, right? With a big C statement that lists a bunch of cases and in each case dials in a gain. That's completely standard. Okay.

But, in fact, again, if you think past the 1969 mentality here's what you would see. The MPC problem is highly structured, so there's a whole section on this, but the Hessian is block diagonal. I'm not gonna go into the details. The quality constraints are block banded. It's weird because the equality constraint matrix isn't square, so it's not clear what banded means, but if there were a meaning for banded square matrix this would be it. But, actually, you know why you can guess this and all you needed was your intuition to know this? This is one of the things I would hope you would eventually get. Probably by the end of this year you'll have it.

It has to do with going from problem structure in the actual problem to the problem structure in the numerics and what are the implications, so let's just talk about that for a minute. Lets say you're solving an optimal control problem and let me just ask you about the state X(t). What does it connect to? Where do the other variables it's, sort of, directly constrained with or connected to?

Student:T plus one.

Instructor (Stephen Boyd):That's what?

Student:T plus one and T plus –

Instructor (Stephen Boyd):It's X(t) is connected to U(t). Lets say it's connected to U(t)–1 because the previous decision is what landed you in that state and it's connected to X(t)+1 and X(t)-1. That's it. Whenever you can make a statement like that – now, of course, X(t) is related to X(t)+50. Okay? But indirectly because it's got to propagate through all these dynamics and things like that. Otherwise, it's just a separate problem, but the key is whenever you can identify blocks of variables and say that – in fact, what we just argued means it's banded. If you stack X and U and X and U(t) you only depend kind of on X and U(t) the previous and post that smells like banded. That says banded.

I mean, you have to go through and work out all the matrices and all that, but the point is at least you know what you're looking for and you're looking for banded and what does banded mean about solving it? It means blazing fast. It means linear in the time step. Okay? So that's the other thing you should know. Banded – how fast does it take a Newton step if you have a band of 20 and it's a million long?

Student:NK squared.

Instructor (Stephen Boyd):Hmm?

Student:NK squared times –

Instructor (Stephen Boyd):Yeah, yeah, time is the answer. Super-fast. And it's not easy solving million variables, million equations generally. Okay? I just want to talk about the ideas. You should be used to this. Structure comes up other ways. In image processing what structure comes up? What are pixels related to typically?

Student: The surrounding.

Instructor (Stephen Boyd):Yeah, some neighbors. Okay. So did you get banded?

Student: Yes.

Instructor (Stephen Boyd):No. You get banded across a line, but you're also related, so you get something called the chronicle product of banded. You're related to your left and right neighbors for sure. Things like signal processing control, anything with time in it, banded. Images you get actually a chronicle product of band. You get weird bands because you're related to the guy to your left, the guy to your right, but you're also related to your neighbor above and your neighbor below. Yeah, okay. Make your more complicated nine point. That hardly changes anything, but it means that you're related here and that's, by the way, an offset of N, right?

So the sparsity pattern there you would expect to see would be bands. Actually, it's a banded matrix of banded matrices. By the way, it doesn't matter. The main thing you should recognize is structure. That's structure and that means fast if you know what you’re doing. Okay. So you can actually solve this. I guess we’ve gone all over this. I don’t have to tell you this. It’s t(n+m)3. So to do this it’s linear in T. Here’s a fast MPC. In fact, it was just put together by Young. We’ve all ready talked about some of the things here. First of all, you don’t have to do a good job in the planning, so you might as well do an interior point method and fix the barrier parameter and then you can use warm start, which I think – who asked? You asked about that, right?

So you could do warm start and then it turns out you only need to do like two Newton steps or something like that. You can limit the number of Newton steps and this is just a simple C implementation, not even optimized, and you can compile it with whatever it was, O3 or whatever it’s called. These would be the numbers involved here. Here would be one. Here’s a problem with, I don’t know, four states, two inputs, a time step of ten, and that goes down in 300 microseconds. You can have a problem with 30 states, eight inputs, 30 horizon. That’s a problem with 1,000 variables and 3,000 constraints. That goes down in 23 milliseconds. Okay?

So I don’t mind – I mean, I just came back from Zurich where there’s people who have been doing MPC for a long time and those who had the 1969 mentality successfully convinced other people that solving a QP is a big deal. By the way, these are also the same kind of people who just used tools and don’t know how they worked inside. If you fire this up in CVX you’ll get actually some very little CVX overhead time. You’ll get 3.4 seconds. All right. This is actually why I think it’s very important if you’re gonna use things like convex optimization. Maybe you’ll go into an area where it doesn’t matter, and you can use high-level tools and you can use other people’s solvers. Good for you. Very good for taste. Very good judgment to go into a field like that.

It requires a combination of modest number of variables and no real-time constraints. If you’re considering fields go into a field like that. However, the point is if you go into something that is huge you’re gonna have to write your own solver. We went over that a couple of weeks ago. If you do anything that’s real-time you’re also gonna have to write your own, but it’s really easy. I mean, it is not hard provided you know what you’re doing. If you’ve taken 364a and b you know what you’re doing hopefully. You should know what you’re doing or something like that. By the way, I do want to make a big deal over this. There’s not a lot of people outside this room who actually know these numbers. Even though everybody should know them, I know all the experts who write all the big codes, if I asked them how fast would you solve – actually, no one even thinks about solving small problems.

So even the experts in the world when you ask them how fast can you solve that? They’ll say a couple of seconds. Right there. That won’t help you if you’re controlling a robot or something like that or doing navigation updates or whatever it is. I personally happen to think there’s a lot of opportunity for MPC running at rates like 50 hertz, 60 hertz, and so on. By the way, there’s other areas where this comes up. There’s a student who did a project in 364 three, four years ago that had to do with optimizing guard bands and wireless transmission. He used an SOCP with, I don’t know how many variables, 200 variables, and during his project he got a C program to run it down to 500 microseconds and then it ultimately went to hardware and it was right in there at packet time. It was in the microsecond range. I forget what it was.

These real-time things are actually quite interesting and fun. I think, actually, they're gonna have a lot of impact. Okay. So lets look at examples. This example essentially is supply chain management because it's completely generic, so here's what it is. I have a bunch of nodes and these are warehouses or buffers, it depends on if you're storing goods or storing packets or something like that, for fluid or whatever. This is gonna be single commodity. It's very easy to make this once you understand how to do this. Totally trivial to make it multi commodity.

Then you have m links that go between the nodes in the warehouses. These could be communication links. They could be airfreight or something like that. It doesn't matter. Pipes. Xi(t) is gonna be the amount of commodity at node i. In period t, uj is the amount of commodity transport along the link j. Then you will split the traditional graph node incident matrix into its negative and positive parts and we'll call those Ain and Aout and this basically tells you whether link j enters or exits node one.

It's something like Ain minus Aout would be the traditional graph node incidence matrix. That's the one that's got the one and minus one on every edge. The one tells you what it points in to and the minus one tells you where it comes from or the other way around depending on which standard you use. The dynamics is this. It's linear. It says that the amount you have at a certain time is equal to x(t) plus this is the amount that came in and then minus that's the amount that flowed out. And let me just for fun ask a question. Suppose you had spoilage. In fact, suppose I told you that this is a perishable good and at each step 10 percent of the commodity goes bad. Please modify the equations.

Student:[Inaudible].

Instructor (Stephen Boyd):Yeah. Thank you. Would it be linear then still? Yeah, sure. Okay. I'm just pointing out there's a lot of stuff you could do. I guess you did that, right? In your thing. If you have somewhat spoilage in airplanes that's not good.

Student: No.

Instructor (Stephen Boyd):No, okay, okay. All right. All right. Here are the constraints and objective. These are just sort of generic. One is to say that the X is the amounts go between zero and some x max. You may not have to have this x max. That would be a buffer limit or something like that. By the way, it's also very common in some cases to let x go negative. In which case, it represents a backlog. It's the same ideas having a negative investment in asset. It means you have the obligation to produce it or something like that. If a customer makes an order you can actually take the order and now you have a backlog, which just says how many units do you have on stock. You can say -12 means 12 are backordered. I mean, you're gonna have to add a penalty for backlog if you do that. You have to add a steep penalty for backlog, but that's another version of this.

Link capacities go between zero and some maximum value. This is a vector of the amount removed from each node or warehouse and that's gotta be less than x(t). Some people call this actually a – do you remember what they? In fact, I just saw some presentation where this had some name to people who do supply chains. It means that what you can't do is you can't ship ten units to a warehouse and then ship it out on the same time period. Everybody know what I mean? So if you didn't have this it would mean you could actually do it all. Like a digital circuit, like a two-face clock, or something like that basically. On the rising edge all the trucks leave and this is that.

This says the truck can't leave with stuff that's not there and then on the following edge of that clock they all arrive. That's my model of it. Okay. You can have a shipping transportation cost that's S(u(t)). This could be positive, negative. It can include sales revenue, manufacturing cost, all sorts of stuff in it. Whereas a storage cost is just a charge on W. If you allow x's to go negative, you would have, in fact, a back order charge, which could be real or just an internal accounting. Your objective would be to minimize this. I should mention one thing. There are some very common shipment and storage cost charges, which are not convex. That would be like a fixed charge. We're not gonna look at those, although you could.

They would be things like this. The cost to run a truck from here to there is fixed. So if you're gonna put two little boxes in the truck it's gonna cost you the same as filling it. Okay? That would have a cost that would go up like that and then, actually, it would jump up again. These would be highly non-convex. Obviously every time you jump from zero to one, one to two or trucks or something like that. We're not gonna look at those, although one could. Now we'll just look at an example. It's an example with five nodes, nine links here, and you start with everybody has some amount of commodity at different nodes. Presumably, this is sold or something like that, so we'll get revenue from shipping stuff out on these two links.

At each step I can ship whatever I like along these edges. I mean, it has to be there for me to ship it somewhere. I'll have a buffer limit of one. The most I can ship out is .05. My storage cost will look like this. It's linear plus a quadratic term. The shipping cost will just be linear. I pay linearly to ship among warehouses. Now, my shipping cost, I don't know if it's fair to call it shipping, but when I pull things out of those bottom two nodes I actually get paid for it. That's why this is negative. That's revenue from delivering product here. The initial stock is 10011 and we'll just run MPC with five steps or something like that. The optimal cost, by the way, is 68.2 and the MPC cost is 68.9, so, once again, just five steps is enough planning horizon to actually do very well.

And lets just see what this looks like here. What I'll do, I guess, is I'll start shipping these out immediately. That's clear. But then I won't have much space here, so I'll actually spread the product around and then ship it. Actually, the point is even baby examples like this what to do? Not obvious. Not remotely obvious. And you can imagine what would happen if you have 40 links and 50 and it goes on and on. Okay. So here's the example.

So the optimal was obtained just by solving a very long problem and this just shows you a couple of things. I guess the first one shows you – this is X1, which starts at one. You can see stuff is shipped out, then held, and then it's shipped down until about 30. This is X3. It goes from zero. It goes up and then down again. Sorry, that's – well, it's the same thing. There's the optimal and there's the MPC. They're not quite the same, but they're very close and you can see here these are the shipments made. The optimal says that, for example, U3 should ship a bunch then, sort of, stop, something like that, and then U4 should ship nothing, and then somewhere around 12 start shipping stuff out like that.

And then this is the MPC one. You can see, actually, it's not the same, but it's good enough to give you within two percent the optimal cost. In fact, if the horizon were any longer they'd just be indistinguishable, the two. By the way, just the pictures of these tell you this is not obvious. This is not remotely obvious what to do here. Maybe we'll make a movie. Did you make a movie? I can't remember.

Student: Yeah, I did, but it's not for –

Instructor (Stephen Boyd):Yeah. Presumably, you could make a movie or something that would kind of show you the right thing to do where after several steps you'd spread your product all around and then after one thing is done the stuff starts pumping out the bottom or something like that and that minimizes your various costs. Okay. Let me just mention a couple variations on the optimal control problem. We've all ready said this. You can have time varying costs, dynamics, and constraints.

One very extremely common thing, especially in finance, is discounted cost where the cost in the future is actually discounted by a discount rate, like you divide by 1.05 to the t in the future. What's funny about that is the people who do control have discount costs with negative, right? They actually put higher cost on the future, which induces the control to be more aggressive to bring something to zero faster. No one in control would ever, ever use a discount cost and everyone in finance would use a discount cost. If the whole thing is going down over a long enough period. We've all ready seen this. You can have a couple stay input constraints. You can add slew rate constraints. I'll just ask you one question, then we'll quit for today, and then that's our next topic, but let me ask you this.

If I added constraints like UT+1-U(t) is less than that it's a slew rate constraint. It says that your input can't change by more than some amount. Tell me what that does to the problem structure.

Student:[Inaudible].

Instructor (Stephen Boyd):It what? It smoothes out the solution, sure, because it says that you can't change very radically. But what did it do to the problem structure that you need to solve? It preserves locality in time. In other words, it's another constraint, but it affects only things close in time. That should go in your brain. The whole center of your brain that controls sparsity patterns and things like that should be lighting up like banded,

banded, banded, banded, banded, and the other one should be going like fast, fast, fast, fast. Okay. We’ll quit for today.

[End of Audio]

Duration: 81 minutes

ConvexOptimizationII-Lecture17

Instructor (Stephen Boyd):Hey. Today we're going to do our, I guess second – second to last topic. It's one of the new ones we were very happy to add. It's stochastic model predictive control, and although in the end it's going to look just like the model predictive control, it's actually worth going over this carefully because there's actually some real subtleties here.

This is actually much more sophisticated, in fact, than mere optimization. And I'll make that clearer as we go along with it. A lot of times, the subtleties are brushed over. I think in this case it's a serious mistake because it's really – it's quite complicated, and you really have to kinda be on your toes to know what things mean and stuff like that, so – okay. Although, in the end you'll find out what do you do about it? You use, like, model predictive control that we looked at last time. So in some ways you could argue that you don't need to do model predictive control unless it's a stochastic problem, so.

So let's – I'm gonna go over this slowly because it's actually a bit – it's quite subtle, in fact. So here's what we have. We have a linear dynamical system, so with a state Xt and an input Ut. And we could make the – we could easily make this time varying so that could be AT and VT. It's easy enough. And here's the new element – is a noise here – a process noise.

So what this is is that the next state depends on actually two things – well, three things really. It depends on the current state, so that's this. It depends on your action, and it depends on this random variable. Actually, at this point, it's not yet a random variable, but it's something – the idea is it will soon be a random variable – and it depends on something that you don't know fully. That's W of t.

So we'll use this notation that capital Xt is gonna be X zero up to Xt, and that's the entire state history up to time t. So it's a big vector with all the states. It's the first part of the state trajectory here.

Now when you talk about control or something like that, we're gonna make the requirement that the input must be causal state feedback. What that means is this, that this input Ut is a function of the state history.

Now, if you're a function of the state history, then you're also a function of the disturbance history, okay, because if I know the state, if I know the state sequence – that's what this says – if I know X zero, X one up to Xt here, then up here I know that. I know that. So I subtract those. I definitely know that because I did it. So I know W of t.

So in fact – in fact, this is just a linear transformation that maps – in fact, a linear bijection that would map X zero up to Xt to X zero up to W t minus one. So to say that you are – or to use the language of probability – to say that you're measurable on capital Xt on the state sequence means you're measurable on the disturbance sequence – actually up to t minus one because you know what the last disturbance is, but you don't know what the next one, or current one is gonna be, or something like that.

Okay, now these functions are called – that's called the policy at time t. Then I guess you'd talk about the policy would be a collection of these functions, or something like that. We're just going to consider the finite horizon one. You could easily consider the infinite horizon version.

And there's a couple of important things to say about this. The distinction in something like stochastic control is that you're actually – what you're actually looking for are these functions five t. In particular, you are not looking for a particular U. In fact, it doesn't even make any sense to say that you're looking for U. I mean, in the end, of course, what'll happen is we're going to find U. That's true. But in fact, what you're really looking for is this U that depends on the state history.

And let me just say a couple of things about how this – why this is extremely different. It's different this way, and it's very important to kinda understand these distinctions because these are not, sorta, mathematical subtleties that don't really matter much. These make all the difference. They make a huge difference in practice, and here there are.

It's this: the basic idea here is that Ut, you actually have recourse. When you choose U three, you know what X three is. You know what X two, and X one, and X zero are, and you only decide U three at that point. Okay. So that's the – in other words, you wait until information is available. What do you know at, for example, t equals three that you didn't know at t equals zero? You know a lot. You know W zero, W one, and W two. So these are things you know at that point. And that's by virtue of here, of knowing the X sequence so you know the W sequence – so you know the initial state.

So that's actually gonna have a real difference. In fact, if you choose U ahead of time knowing nothing, you will do worse. You must do worse, of course, and the difference actually is gonna be sort of the value of recourse – recourse, or feedback, or whatever you wanna call it.

Okay, so let's look at stochastic control. That works like this. The initial state and the sequence of disturbances is a random variable, though we don't assume that they're independent or anything like that yet. We may later, occasionally. So, this is a random variable. Well, it's a stochastic process, although that's a little bit pedantic, to call it a stochastic process, since we're doing finite horizon. But if you want to call it a stochastic process, go right ahead.

And our objective is going to be this. It looks exactly like it did last lecture, so it's gonna be a stage cost. That's little lt of XtUt. And then there'll be a final state cost. So this is sort of the – this is the cost of winding up, or whatever – something like that in one of these things. In a supply chain that would be some salvage cost, or some who knows what, that kind of thing. Okay.

In a dynamic system problem, this might – in fact, the whole thing might – you might be interested in putting it in some final states, in which case this would be less important, and this would judge how far you are from some target state that you wanted to hit, for example.

Okay. So [inaudible] these are convex. And what's interesting about this is that J – this objective – now actually depends on these control policies. So these are really our variables, if you wanna talk about variables in the problem. So this is an infinite dimensional problem, and in fact, in some sense it's even more than merely like an infinite dimensional control problem because your variables are actually functions. So even if you do one stage, or whatever, that has a name – if you just do one stage, your variables – I mean, if there are variables here – there are – are these actual functions or policies.

By the way, that has nothing whatsoever to do with how we're gonna implement it. They implement it be actually solving specific problems, or working out functions, and all that kind of stuff. That's a different story. Here, what's important to serve is what depends on what and when decisions are made.

Okay, and we'll have constraints. Ut is in this script Ut, and the stochastic and full problem is this: choose these policies of phi zero up to phi t minus one to minimize J, subject to the constraints. So no matter how you write it down, that is actually a very complex problem, and it's worth actually thinking about what it means.

It is not the same as an optimal control problem. An optimal control problem basically says, "You're in this state. Here's your cost. Find me a sequence of inputs." That's a straight optimization. Maybe it's large dimensional.

Even in the infinite dimensional case, which we looked at last time, it's not that big a deal. I mean, you just take a finite horizon and that'll approximate it well enough. This is much more than that because you're actually optimizing over – in fact, not only are you optimizing over sort of these really infinite dimensional function spaces, but in fact this growing set.

So, I mean, it's not that big a deal, but it's very important to understand what this – this is not – what you'll find, actually amazingly – and a lot of things is it just says the problem is to choose U to minimize J. And that's actually just, like, wrong because that's not the problem. It makes no sense to choose U. What you're actually choosing is you're choosing the method by which you choose U when you've arrived at a certain point in time, and you've made observations. That's actually what you're choosing, which is a policy.

Okay, so as I said, that's an infinite dimensional problem, and the variables are these functions, or policies. And there's many things you can do. For example, you can restrict these policies to a finite dimensional subspace. You can say all these policies are affine, for example. You can make it even more specific than that, and a lot of classical control stuff that you learn about in, like, a control class or whatever, would be where these – they don't make a big deal about all of this, and these are very specific. You might like at some error, and have something proportional to the error, to the running sum of the errors, and maybe to the first difference. That would be a classical PID control.

By the way, if that's the type of control you're doing, where you're looking at the extremely simple policies like that, then it's pedantic to even introduce all this and talk about policies and things like that, but that's – okay – but we're gonna look at the sophisticate – the full case here.

Now, the very important point here is that you actually have recourse. Recourse means you can actually come back and do something about something after you learn more information. So that's what recourse means. It always gives you an advantage. It basically says – another name for this is, like – do I have it here? I don't. So multi – well, there's lots of names for this. One is called multistage optimization, sequential decision making. Actually, those things should be up there. Those are other names for this type of thing, and they all kinda make sense. Oh, the other one – the very big one – is just feedback, so.

I guess you're taught about this as an undergraduate. I taught undergraduate classes on this, and you give – you talk about feedback. I guess these ideas come up, but not the – not really fully what it means in an optimization context. And this is what it means.

Okay, so the point here is that you actually get to change U based on the observed state history. So if you look at a standard optimal control problem it says basically, "Here's the problem data. Please decide on what the state – what the input history should be to minimize the expected cost." That's a stochastic – sorry – that's a stochastic optimization problem. So even if there's random variables in it, that's no problem. But that's still just an optimization problem. In fact, in finite horizon one it's a finite dimensional problem. It might have some random variables and be a stochastic optimization problem, which some people would have you think is hard, but it's still just an optimization problem.

So this would be called instead of feedback or closed loop control, this would be called open loop, is what you'd call this. Okay, so now in the general case, the only way to really evaluate this objective here is gonna be here.

There's gonna be some special cases. These are quadratic, and so on. We're gonna get a sort of analytical solution, and, in fact, as usual, that will be essentially the only – that and 15 other special cases – will be the only cases where people can solve stochastic and full problems. That will be the – solve meaning like here's the solution. I mean, there'll be isolated ones in one dimension, and two, and things like that. We won't even look at them, but there are plenty of them.

Actually, you can also say the following, that if there's – I don't know – if the state dimension is one or two you can certainly solve it numerically, and get as close as you like to the solution, and so on, and so forth, and it's very, very simple, but not in the general case.

Okay, so the only way, really, to evaluate this – except in these very special cases – is by Monte Carlo. The way you do that is you fix your policies. You have some policies, and the policies could be as simple as some affine policy. They could be open loop. You could compare it to an open loop policy. An open loop policy would just – these functions are very simple. Whenever they're called they just return a – they're returning constants. They just return some U. This returns U zero or something. That's what you're gonna do whether – no matter what you observe.

What you'll do is you'll simulate a trajectory of X zero and W, and you will calculate J, and you'll run Monte Carlo. And you can make a distribution of this cost, and then the expected value – in this case the empirical mean – will give you the objective modulo Monte Carlo error. So that's what you'll have to do to judge these.

Okay, so it's the solution, the dynamic programming, so you can actually say what the solution is here – or you can say something about the solution – and it does, in a few cases, actually just give you – it becomes sort of computable. So let's look at what it is.

It works like this. You're gonna let Vt of capital Xt be the optimal value of the objective from t on assuming – or conditioned on – the fact that you visited the states X zero up through X little t, which is to say that you have this state history capital Xt. That's your state history up to that point. So conditioned on that, you then let things play out on the other end. You would run, for example, to stimulate that you'd start from point conditioned if this is a distribution, you'd work out the conditional distribution for Wt, Wt plus one, all the way up to Wt, capital T minus one conditioned on this sequence. And then you would do simulations and calculate this optimal value of the objective.

Now, we can certainly say what this at the end. At the end it turns out that Vt, of course, there's no more action to take so there's nothing to do. In fact, there's nothing to do except pay the final bill, so you simply pay the final bill here, like that. That's deterministic. There's no – this is – in other words, this is X zero up to X capital T, and it only depends on this last component here. And if you take this back to zero, then the expected value of V zero of X zero – X zero is your random variable – this is actually exactly equal to the optimal value of the original problem.

I should say something about this before we move on, that you can do all – that you can compare all sorts of other stuff here, and there are actually two extremes that are actually worth pointing out. Let me just point those out now because it's good to keep these in mind here.

So there are two extremes. One is actually the prescient control. Prescient control basically says, "I will tell you ahead of time what the disturbance sequence is." Prescient means you know the future. "I'll tell you what the demands are gonna be, or I'll tell you what disturbances are gonna hit your vehicle," or something like that.

And if you do that, it's interested because now the – because in this case the control – it's not causal, of course. It knows the future. And if you actually solve the prescient version, it still depends on the initial state, but that, by the way, is a standard convex optimization problem, finite dimensional. You can solve it. If you run the prescient version Monte Carlo, starting from different X zeroes you will get the average prescient cost. That's clearly a lower bound here. So that's gonna be a lower bound. Everybody see what I'm saying here? If you – so –

Of course, in some cases knowledge of the future is gonna be a huge advantage. I mean, think finance for example. Even a small amount of knowledge of the future is gonna give you a huge advantage. Actually, in other cases it might not. So that's one down.

Now, at the other extreme – so that's the full prescient version – variation – which gives you a lower bound. You could also do the other one, which is the full ignorance policy.

The full ignorance policy goes like this. You can do it different ways, but the full ignorance policy is this. Choose U zero, U one, up to U capital T. Now, before you know any – any – of the Ws – see what I'm saying? By the way, that's covered by last lecture because that's nothing but – that's an optimal control problem, or what people call optimal control. That's the full ignorance one, and that will, of course, do worse than if you have recourse, and the extreme of recourse is to have full prescience. Okay. So this sort of in between here. Okay.

Okay, so let's get back to the solution here. So you have – this is the optimal value of the objective conditioned on this initial state history capital X sub t. Then you get a backward recursion, and it looks like this. It says – well, in fact, you can even say what the optimal thing to do is. In fact that's the first thing to do.

What you do is this. You have observed capital Xt. So what you do is this, is you should minimize the current cost of a current action – that's this. That's the bill you're gonna run up now. You know Xt because it's conditioned on capital Xt. You know little xt. For that matter, you know all the – the entire previous state history. And then what's gonna happen is as a result of your action V, you're gonna land in this state. AXt plus BV plus Wt. That's random because you don't know Wt yet because you don't know where you're gonna land.

Okay, so what you would – wherever you land, you'll have to finish out what you're doing. And if you do this optimally, the cost will be V t plus one of where you land here. Okay? So – and this is the state appended at the end of this thing.

So that's the cost of where you land, and this will be conditioned on – you have the expected value of this conditioned on Xt. In fact, this expected value goes outside here, but this thing is deterministic if you condition on capital Xt so it goes over here. Okay? So this is the picture. This is the so-called – I don't know – the Hamilton-Jacobi equation, the Bellman equation, the dynamic programming equation, and it's got a Russian name, which I don't know. So, anyway, that's what it is. Does anyone know the Russian name? Pontrialigan, let's say.

Okay, so then you get this backward recursion. It's exactly the same as before, except we didn't have this conditional expectation before, so this was not here. This was simply simpler, and this was not here either. So what happened in the last lecture was this. When you chose an action, where you ended up was absolutely certain. It was simply AXT+BV, and you simply had to worry about balancing the current bill versus the bill you run up from landing in a certain state at the next step. Here it's more complicated because even when you commit to an action, you don't know where you're gonna land next step, and that's why you have this conditional expectation here.

Okay, now, you can show that these are convex functions. That's actually quite straightforward because any kind of averaging or conditional expectation is gonna preserve convexity. And this is minimizing. If you do partial minimization, you preserve convexity. So these are convex, these functions.

Okay, an important special case is when you have independent process noise. So that's when the noises are independent. By the way, that doesn't happen that often. Typically what happens is you transform the original problem so that the noises are independent. You do that by introducing additional states.

Actually, if you've taken these classes you know about this. But let me just give an example of – let's say in finance – a very common model of something would be a random lock. In a random lock, obviously the variables themselves are not independent. What's independent are the increments, are the differences. So you'd introduce a new variable, which is the running sum of the Ws. That would be a new state. So that's how this is done.

Actually, how many people have seen this somewhere? In which kind of classes? MS&E or –?

Student:[Inaudible] dynamics program.

Instructor (Stephen Boyd):Oh, okay. Perfect. Okay, great. How about –?

Student: Yeah, me too.

Instructor (Stephen Boyd):Same?

Student:MS&E 251.

Instructor (Stephen Boyd):Cool. Okay. All right. No one in [inaudible] 210? Did you take it?

Student:I took it a long time ago.

Student: Haven't taught 210 in years.

Instructor (Stephen Boyd):Oh. Well, okay. Fine, but had they taught it, this might have been in it.

Student: You see a little bit in 207 A or B, one of those.

Instructor (Stephen Boyd):B.

Student:B. Yes.

Instructor (Stephen Boyd):Okay. I thought it was in that one. Okay. Really? Years? Oh, well. All right. So, okay.

So this one actually gives you a solution in the typical case. I mean, actually, after a while you get used to this. If everything's like Gaussian – actually here you don't need anything Gaussian because it's just second order statistics, but with enough Gaussian independent arrival, exponential arrival, these kinds of assumptions of linearity, and quadratic functions, usually these problems get solvable, or something like that.

Okay, so we'll assume that here X zero, W zero – actually, let me go back because there's something I didn't – I didn't finish saying something about this, which I meant to say. Sorry.

In this case, when you have an independent process noise – again, an example of how you'd create an independent process noise in the general case – then Vt actually only depends on the current state. This is what you would – this is the way things simplify, and they look much simpler. So V sub t of capital X sub t you can write as V sub t of little xt, because that's all it depends on. And then you get this, and this is actually what you would see in books, more likely. You'd see something like that in books. And then the optimal policy would look like this.

By the way, there's an interesting thing here. The – because of this expected value, this actually has a smoothing effect on Vt. If you have a function V, even if it's sort of non-differentiable, right, [inaudible] corners, and things like that – if it's like an L one norm – you throw in an expected value, and that actually is gonna smooth it out. So these actually are gonna be quite smooth functions. That's actually a smoothing effect.

Okay, so let's do linear quadratic stochastic control, so no constraints. These variables are independent with zero mean and covariance is sigma and then Wt, something like that. That would be sort of the standard – a standard model. And then we'd take stage and final costs which are just quadratic – convex quadratic. And you don't really need this Rt to be positive definite, but it's a traditional assumption, and it makes sure – it means there's an equation that's going to come up. Well, the Bellman equation is a formula for it that just always works. The important point is that these are quadratic – these are convex and quadratic.

In this case, the value function itself is quadratic. It, by the way, is not a quadratic form. So it's not – it's a quadratic function so it's got – it's got a constant. Actually, it doesn't have the linear part. Although, if your original problem had a linear objective – a linear term of the objective, this would have a linear part, too. At the end, you have Pt is Qt. That's the final cost, and that little one is zero. And then Vt, you have a backward recursion from here. This is – this thing is V t plus one of AZ plus BV plus W. That's what this thing is.

Now, when you work this out, it's quite easy to do here because lots of things are zero mean. So, for example, the Wt transposed P times all of these two, that goes away because that's zero mean. And the only thing that really comes up here is you have the Wt transposed P t plus one W of t, and that's this. That you would recognize this way. Right, because the expected value of Wt transposed P t plus one W of t is that. Right? People have seen this before? This is the usual.

Let me just write that down just to make sure it's completely clear. It's this – it's a completely standard trig. It's sort of like if you wanna write the expected value of Z is – well, actually all I need is this. All I need is this. If I have expected value of Z equals zero, and expected value of ZZ transposed equals capital Z, then the expected value of Z transposed PZ, you write it this way. The first thing you do is you rewrite this thing as the trace of P times ZZ transposed, which looks kind of pedantic and weird here.

You do that – by the way, the way you know that this is right is that trace AB is trace BA. So I could write this as trace PZ times Z transposed is the same as trace of Z transposed times PZ. But then it's trace of a scalar, so it's the same thing. But this thing, the expectation floats inside to the ZZ transpose, you get capital Z, and you get the expected value of PZ like that.

By the way, that's sort of – people write it down that way. That's perfectly good. Everyone understands what this is, but probably there's better ways to write it. For example, I think this is probably a better way – which would be one better way. So that's another way to – that's a pretty symmetric form. So this would be a better way. It won't matter, but it's for aesthetics.

Okay, so this term Wt transposed P t plus one Wt expected value, that turns out to be this thing here, and the rest turns into a familiar Racadi recursion here because here you just get some quadratic in Z and V. You minimize the quadratic in two blocks of variables over one, and you get a sure compliment.

So this is actually a sure compliment, although you can't really see it that well. You would see it if you pulled out – I don't know – certainly if you pull out an A transpose out of either side. Actually A transpose would be fine here. You pull out an A transpose on either side, and a sure compliment will emerge here for something.

Okay, so that's the – what's nice about this is this gives a completely – this is the one case where you can actually solve a stochastic control problem. It's quite sophisticated actually. The optimal policy is the linear state feedback. So in other words it just takes the current state, multiplies it by state feedback gain, which you get by propagating this Racadi equation – this Racadi recursion – backwards in time.

The first thing you notice about the state feedback is that it actually apparently has nothing to do with the statistics of what is gonna – of what you're up against – I mean, the disturbance, and also the initial state, which seems kinda weird. So it says basically that you do the same – you take the same recourse no matter what the – no matter what the – or the policy is the same. It doesn't depend on what kind of disturbance you're up against, or the covariance of it, which is a bit odd at first.

And the optimal cost is the expected value of V zero of X zero, but that's trace sigma zero of Q zero, but if you go back and look what Q zero is, Q zero is nothing but – you can see that Qt is just a running sum of these things. And so Q zero is simply the sum of these things, and that's – that gives you this formula here.

Now, this gives a – all of this can be interpreted quite nicely. This is exactly the optimal controls we looked at last lecture, but that was not a – that was an optimal control problem, not a stochastic control problem. It just said, "Find me the optimal U to minimize this cost," and all that sort of stuff, given what state you were in. And there was – it was a mere observation that you could write the optimal controller in a linear feedback form, and it was utterly irrelevant in the last lecture. You could give U as a trajectory. Immediately you could write it in state feedback form. They were exactly the same.

Here, that's not true. It is very important here because the point is that now you don't get – you don't know what X little xt is until time t, and that's when you make your decision as to what Ut is, and it still has its linear form.

So these actually have some very nice interpretations. Let me see if we wrote them down. I didn't. Let me tell you what the interpretations of these terms are. This one is quite beautiful. This one is it says this: it says that, "I'm going to choose at random an X zero from the distribution of X zero. I'll choose that X zero. Then after that there will be no more disturbances, none. So I'll just start you in a weird initial state, and I'll ask you to maneuver that – X zero – not quite X zero – whatever minimizes the cost running forward."

If you do that that is exactly the average costs you run up. That's what this term is. So this term is the cost you would get if the initial condition were generated according to this distribution, but there was no further disturbance of the state. In which case, by the way, you could work out what the optimal U is as a function of the initial state. So that's exactly what this term is.

This term is actually quite beautiful. It basically says that – it's the same thing, but moving forward it says, for example, at time three – or something like that – it says, "Imagine that you start from an initial state depending on – which is chosen for the distribution W three, but thereafter, there’s no more disturbance.” So that would give you these. And it basically says that the sum of these gives you the optimal cost for this thing. And that’s not remotely obvious at all. In fact, when you kinda look at it, and think these two are the same, it’s just some weird accident having to do with linearity, and quadratic costs, and things like that. It’s certainly completely false, in general, that this would be the case.

Okay, well now I can tell you what – how people – what are the methods for solving these problems. One, maybe it’s the simplest one – it’s not the simplest one – I mean, methods for stochastic control are – well, for one thing it includes all of traditional control like proportional plus integral – proportional integral derivative – all that stuff applies, but a simple sophisticated – the simplest of the sophisticated methods would be certainty equivalent for model predictive control.

It works like this. It’s just model predictive control, but it’s gonna work like this. It’s time t. You have observed the state trajectory. That’s utterly irrelevant up here, except in one term only. You’ve observed a state trajectory zero, X zero, X one up to X little t. Based on that, you form a prediction of what the future disturbances are gonna be. Is there independence? That’s just like the mean, or something. It could be.

By the way, these absolutely do not have to be anything like conditional expectations. So they don’t have to be. They can be conditional expectations. By the way, that’s no better or worse because, in fact, the optimal thing to do is not to use conditional expectations here, but the point is they don’t have to be conditional expectations. They can be. If you can calculate the conditional expectation, great.

These could actually be – these are just forecasts. I mean, they can come from analysts who say, “I think the prices are gonna look like this,” or something like that. It just doesn’t make any difference. So this is some prediction of what W tau is gonna be.

And so what you do is you form this optimal control problem. Now, as of – like last lecture – and you’re gonna run out capital T steps – sorry – you’re going to run from where you are to the end of the game. You’ll run up – you’ll do the accounting for your stage costs and your final cost. You’ll require – you have to satisfy the input constraints. And what you’ll do is for the disturbance you’re simply gonna put in these forecasts or approximations. You’re just gonna plug in the forecast.

It’s really, like, quite stupid. It basically says – it says, “Do a planning exercise. Do exactly what you would do if your forecast were perfect.” Right, which is quite ridiculous. So, but that’s what it is.

So the only way in which capital Xt comes into this – and in fact, this whole thing is actually – this is the policy, the Arg min of this subject to that – that’s a policy, which is to say it’s a function of capital Xt, and it comes in right through there and nowhere else. So you solve this problem.

Okay, so you called it – this is your plan, and it's a bit silly. These are all – actually, not one of these will actually come true. I mean, this is sort of your – if someone says, "What's X little t plus one out to the end?" You say, "Well, that's my plan for what the state trajectory is going to do." You have no reason to believe that's actually what's gonna happen. In fact, that's not what's gonna happen. That's all quite – in fact, if that's what's gonna happen, that means your forecasts are perfect, and this is the wrong lecture for you. You have to look at the previous one, which is simpler because then it's not stochastic control.

Okay, so you work this out. Then what you're gonna do is you're simply gonna do this. And then this is – that's your policy. That is the certainty equivalent model predictive control. That's your policy.

So it looks dumb. It works actually shockingly well in a lot of cases. It sort of balances two – it makes two simplifying assumptions which kinda cancel – null each other out. One is that it assumes your forecasts are perfect. The second is that you'll have no recourse in the future. Actually, if your forecasts are perfect, there's no need for recourse. So this combination of two weird, bad assumptions kind of ends up doing something quite good.

Okay, so this is widely used, for example in revenue management. This is a euphemism for extracting as much money as possible from you and me, like the airlines use, or something like that. They decide how many seats to release at some price, and that kind of thing. So that's revenue management. I like that term. They're definitely managing the revenue. And I'll give you a hint. It's not in our favor.

It's used actually in all sorts – it's also used in finance for a lot of things, for things like optimal execution and all sorts of other stuff, when you have to sell a giant block of asset, or pick up a block – a big block – some big block of stock, or something like that. So it's used in lots and lots of things. In fact, it will be used in more and more. It's – right now it's used in things where things go pretty slowly. I mean, where it's down to a second, or even minutes, or something like that – like in supply chain optimization.

In real supply chain optimization, that's stochastic control. Unless your forecast says, "I'm gonna tell you the demand for this product in three weeks," and then you say, "Well, how sure are you?" and then the forecaster says, "Totally," then you're in last lecture.

By the way, the funny part is you'd do the same thing. The same code would run. So it's really just subtleties here. The difference is in that case it's just a cheap way of just looking ahead and getting out of solving a big problem, or something like that, which is actually kind of silly given that the cost of solving problems like this scales linearly with the horizon size.

Okay, so it's used in revenue management. As I said, it's already – it's based on two very bad approximations that kind of cancel each other out in a weird way. So you trust your forecast perfectly, and you have no recourse. Actually, if you trust your forecast perfectly, there's no need for any recourse because it means that the future is completely known, and therefore you can decide. There's no advantage to changing how much of some product you order next Tuesday based on a week's worth of new data because your forecast is perfect, so there's no need to. And it works really, really well.

By the way, combined with the stuff we looked at last week, which tells you that these things can actually run very faster – faster by, let's say, a factor of 1,000, or 100, than anyone – than almost all people know or suspect. It tells you that there's actually plenty of opportunities for this stuff to be fielded in various things because not many people – I mean, there are places using this. One of the problems is when you looked around and said, "People don't use this name."

The only name – the only people that would recognize model control would maybe be people running chemical process plants, and you said we're using – that's how you solve your stochastic control problem, they would say, "What's that?" And if you looked in finance or revenue management, they would say, "I'm doing revenue management," or, "I'm doing dynamic linear programming," is another name for it, or something like that. So you get – okay.

So let's look at an example. It's a system with three states, two inputs – I mean, none of this matters – a horizon of 50. It's a completely random system. You'll have a quadratic stage cost. This is just – the point is just made up completely just to see what happens – and the constraint set is – it says that each set, that each input has to be between plus or minus point five. Then we'll just make all of these inputs initial state. It's all – they're all just IID with a standard deviation of point five. They're all plus or minus point five variables, all of them.

Okay, so here's a sample trajectory. A sample trace of X one and U one, and this shows you the state, and here's the input. You can see that it is actually saturated some fraction of the time here. You can see various times where it hits the limit like this. And this is running – this is with the MPC trajectory.

Now here, the way you'd get this is you'd generate a random X zero, W trajectory. You'd then actually run MPC, which is to say that you'd solve a QP for every one of these points, and you'd propagate it forward, and get – by the way, the – what's our forecast for our Ws? What's W hat of t given tau – tau given t? What is it?

Student: It's zero.

Instructor (Stephen Boyd):It's zero. Yeah, it's zero because they're independent, and yeah. By the way, if for some reason you don't want it to be zero, you're perfectly welcome to make it non-zero. If you wanna add a hunch in there, you're welcome to. But we're using zero.

So the way you judge this now is it's – unlike last lecture you can't just run it and say, "Here's the objective value." You actually have to do this by Monte Carlo. There's no other way. Unless – I mean, if there are no constraints and it's – then you get a formula, that's the trivial case. In a case like this, just with – all it takes is input constraints to make this – there's no analytical solution in this case. There are a few problems with analytical solutions, but –

And so you run Monte Carlo simulations. Each Monte Carlo simulation requires you to solve 50 QPs in this case, or whatever. So you run – I don't know –a couple hundred or thousand Monte Carlo simulations. In fact, that looks like 1,000. Is that the right number? Is 1,000?

Student:[Inaudible].

Instructor (Stephen Boyd):Really, because that's 100 right there. Of course, these numbers are totally irrelevant, right? But if you add up all these you'd find out – you might be right. That's 500?

Student:A thousand.

Instructor (Stephen Boyd):It doesn't matter, 500 or 1,000. Anyway, it's some reasonably appropriate number – and the average value is here. So of course this objective is a random variable. It's got some outliers. It's got some things it did very well here because you've got a very benign input disturbance here.

By the way, as the horizon gets longer, of course, this gets tighter. It doesn't matter. We're just doing the 50-step version. So this is – I'll explain what these two are in just a minute. In fact, I'll explain what all these things are. In fact, let me do that first, and I'll come back.

So here are – several control laws are gonna be checked. We're gonna do this. The stupid control law is to calculate the linear quadratic one. By the way, this is how this would sort of probably work if people implemented it in a dynamical system or something like that. You'd have the state feedback gain, which would tell you U is something or other. If there were no constraints on the input, that would actually – that linear feedback control law would be exactly optimal, period.

Now the problem is when you form KtXt, occasionally that's outside the bounds of your input, and so you simply saturate that. So that would be this one, and this gives you an average cost of 275.

If you run MPC here, your average is 224. So you can see it's actually measurably different. I think you can see that from this picture here. This is this very, very simple thing, and this is MPC here. So that's a little disturbing, but that's okay. That's the nature of all this.

Actually, if you cared about this – suppose someone said, “I don’t like all of that compared to that,” what’s the answer to that? How do you respond if someone says that? There is a correct response, and there’s a lot of incorrect responses. What’s a correct one? What’s the correct one, actually?

Well, the correct one goes like this. When this problem started, you agreed to this tossed bunch, the expected value of the cost worked out with this. If, for some reason, you don’t like it when this thing has – when that cost has outliers that are big – then you should not have agreed to that cost function. And, in fact, you shouldn’t have done something like this. You could put a risk averse term, and how would you put risk aversion into this?

We talked about it last time, actually. I think we did. How would you put risk aversion? Just suppose you look at this, and someone says, “Yeah, I’ve run MPC,” and they go, “That’s terrible. There’re lots of times when the cost is high. The stupid saturated controller does better, actually.” What would you do about it? I think we talked about it.

Student: Have like, standard deviation –

Instructor (Stephen Boyd):You have standard deviation; you know how you’d do that in a convex way?

Student:X transpose covariance X.

Instructor (Stephen Boyd):No, that won’t do it. That won’t – because you want – what you wanna do is you wanna penalize large excursions of Lt – L little t and L big T. You wanna penalize large ones, and you’re less worried about the smaller ones, right? It’d be like what you do in finance, or something like that. What would you do?

Student:[Inaudible]

Instructor (Stephen Boyd):Huber would be much worse. If I replaced those quadratics with Huber, I’m basically saying, “I’m giving you license to have large residuals,” and that would do just the other thing. But you’re close, actually. What would be better than that? What’s the opposite of Huber?

Huber says, “Relax on the outliers.” Right, because it just goes linear. That’s the most relaxed you can be and still be convex in a penalty function. Barrier. Barrier would do the trick.

By the way, if I did a Barrier here, what would happen? Actually, can I do a Barrier here? It’s sort of a subtlety, and it would be a practical issue. You cannot because the disturbance is Gaussian, and so with small probability, however big you set your barrier, there’s a positive probability that the disturbance – through nothing you can do about it – would be outside the barrier and run up a bill of infinity.

You can't do Barrier, but you can add, like, a quartic. But I'll show you the right way to do it, in fact. The correct way to do it is to do risk aversion. And you would simply do this. You'd have this thing. I'll – it doesn't matter. Let me just do this one. You don't like excursions in X, then you'd do this. You do – this is your new – that's your new cost. Okay? That's your new cost.

By the way, it does the right thing. If gamma is really small, then this is basically like that. But what this does is it penalizes positive variations more than negative ones. Everybody – this does the right thing, and when you put the expected value here, that's your new cost function. It does just the right thing, and it's very traditional to do this like that. And this would be a cost function that is now risk – it would be called a risk averse quadratic – or exponential of quadratic, and whatever. So that's what you would do.

By the way, the first term in the expansion is what, I think – I can't remember who suggested it – what you wanted. The first term in the expansion here is the variance. So you would get this cost plus the variance. And if you were to do this, it would tighten this up very, very easily. How easily could we do this in –? Well, I believe – is the source code online?

Student:[Inaudible].

Instructor (Stephen Boyd):Not yet. If the source code – how fast could we do this, could we add this, could we modify this – the source code?

Student: Very fast.

Instructor (Stephen Boyd):Very fast, right. You would just go back in, and wherever there's – wherever you see the – you find, you search, you grab for minimum – you find the minimized statement, and you replace it with this thing, and it's done. Right.

So the incorrect answer is to complain about these things. That's wrong. By the way, the same is true, but would actually never happen in finance if you said, "I wanna do something," and then you work it out, and then look at the histogram, and there's some outliers, like in this case negative ones meaning loss, or something like that. You don't like those? No problem. Then go back and put in a nice, strong concave function that charges you for losses. You don't like losses. And don't complain about it. That's the key.

Okay, this is all an aside. So I guess the point here is that when you started the problem, we agreed it's the average of these that matters, and this beats this quite a bit. This relaxed number – let me show you what that is. The relaxed number is simply this: it basically does the following. It ignores the input constraint. So it's simply – it's calculated via the LQR – via the linear quadratic regulator. So it just propagates backwards, and it actually runs up wherever this formula was. It just calculates this formula. That's not by Monte Carlo. That's just – you run through – add up all these guys, and then that's the exact amount.

That’s obviously a lower bound because here the difference between the two is merely that you are more constrained. It simply corresponds to commenting out this constraint, and then you get the global solution because it’s one of those special cases. And that says that no one could possibly do better than this here.

And what this gap shows you is that – well, it kinda says that you’re kinda close. By the way, a more sophisticated relaxation method will make this bound for this instance – what is it – like 175 is a more sophisticated bound. You get 175.

Actually, at that point it’s quite interesting. I would be happy to argue at that point that for all practical purposes this is pretty good. Actually, this would be for the quadratic case. And the reason I would say that is this: if that’s 175, and that’s 225, you’re off by 50, or something like that, but you have to take the square root because these are sorta like squares. So you’re maybe something like – at that point you could argue that you’re no more than 15 percent suboptimal, period. That’s it.

In this case, for problems like this when you see objectives that look like this – this would be typical in dynamic systems with vehicles, or traditional control. Usually, people don’t really mean this. They just mean, “I want this.” This is a code word for, “Please make the state small, not big.” Right, because no one really cares about least squares type things.

If this was a more intricate function in a finance problem, or something like that, or a revenue management problem, then things being like 20 percent within the global optimum is not – then 5 percent means something. So when you see an objective function that’s likely to actually be the revenue, or the loss, or the cost, or something like that, then you can’t be so cavalier about it. But when you see things like this, it’s just somebody saying, “I’d like the state small.”

And you can go back and ask questions about that. Like you can say, “Are you sure you really meant this? Are you sure you didn’t mean one point one here, or something like that?” You can ask a couple of key questions, and they’ll maybe admit it.

Okay, so this finishes up this topic. It’s quite interesting. I think there’s a lot of, sort of, opportunities for doing this in a whole bunch of different areas. I guess people are already doing it, maybe not in a totally organized way, but in lots – people in lots of fields have already found that. Maybe you’ll find it in some others.

Okay, so any questions about this? Otherwise, we’ll go on to our – we’ll start our last topic.

Let’s see. How do you do this?

Okay, so now we’re going to start the last topic for the class, which is on global – it’s the very – it’s sort of your intro lecture to global optimization. That’s, by the way, a huge field. You can and do have whole classes on it. I don’t think here, but you can have whole classes on this. People do have whole classes. There’s like books, and books, and books written on this.

So this is going to be the simplest, simplest, simplest method just to show you what these methods look like – the simplest methods of what it looks like to solve a non-convex problem noneuristically, to be able to say that’s the global solution within this, or something. That’s branch in bound.

So let’s go over how that looks. So just to remind you how this works, we’ve already looked at this earlier when we looked at sequential convex programming. The basic idea is if you have a convex problem, then methods – and you – it’s some standard problem – if you solve it you always get the global solution. That’s not an issue. And it’s always fast. That’s sort of true both in theory and in practice.

When you have a non-convex problem, you have to give up one of these. That’s what you have to do. Now, if you give up the always global – you’re talking about local methods. We’ve already looked at one, which is sequential – or a family of local methods based on sequential convex programming. That’s – these are local methods. There are lots and lots of those that you can take classes on here, entire classes.

In global methods, what you’re gonna do is we’re not gonna drop this. So you’re always gonna get the global solution, and you’re gonna lose this. They do actually, every now and then, in some application they’ll actually work quite well.

Okay, so we’ll look at global optimization methods. They’ll find the global solution, and they’ll certify it, but they’re not always fast. In fact, in many cases they’re slow.

I should say a few things about this. Just to remind you, in convex optimization there’s the idea – there’s the certificate. A certificate of optimality, or sub optimality, or something like that is provided by a dual variable. So you solve the problem. I mean, after a while you get used to it. You just solve the problem, and you trust it. You solve an LP, and you get the answer, and you don’t check to see if, like, STPT three or sudumi calculated the right solution for you. And it wouldn’t have come back with a status of solved, or whatever, in CVX had it not certified that to some reasonable number of digits. It actually calculated an exact – a solution within that tolerance.

So the certificates in convex optimization are very simple. They are dual feasible points. So when I give you back a solution, I say, “Here’s the solution.” And the simplest method of certification goes like this. I’ll give you a point which is suboptimal. You verify that it satisfies the constraints. I also give you a dual feasible point. You verify that it satisfies the dual constraints, and you evaluate the objective of the first one, and the dual objective of the second, and if they’re close, then it means that – in fact, it simultaneously certifies both. So certificates in convex optimization are based on duality.

What's gonna be really interesting is what does a certificate look like for a general, non-convex problem. And there's different kinds of certificates, but we'll see what they are. It's a more complicated object.

Okay, so branch in bound. It's a family of global optimization methods. You'll find lots of variations on this. There's like branch and cut. They've got all sorts – that's another huge name for this, and they have all sorts of names for this.

Actually, how many people have actually taken something on this, or seen –? No one. Okay.

Okay, so they're gonna provide – they're gonna maintain provable lower and upper bounds on the global objective value, just like in convex optimization. So you will get plots that look very familiar. You could even show something called the gap. It's not the duality gap, which you're used to seeing the duality gap versus iteration. It'll be a gap, and the gap will be – the gap between the best point found so far, and the best lower bound found so far. That's gonna be the gap at any given time.

Now, these have exponential worst-case performance, and in fact, they will often be very slow. Actually, occasionally, if you're very lucky, they work well.

I should add that I branch and bound algorithms, you're gonna see like – in solving convex optimization problems, there's not much room for personal expression. I mean, not much. I mean, I can solve it faster than you can, or something. I mean, there is, but it's kind of very small. You get to non-convex problems like local methods, huge room for personal expression.

By the way, that's a way to say these things are totally unreliable, and need to be babysat, and all that kind of stuff. But if you wanna think of it on the positive side, it means that my local method can be way better than yours, which just is weird.

And then you judge it in many ways. You judge it by how fast it is. Also, it's entirely possible in a local method that we come up with different solutions. Sometimes yours is better. Sometimes mine is better. If mine is often better, then mine is better. And I can always, when I'm doing this, choose the examples where mine is better. So that's why I'm saying it's nice for – you get nice personal expression.

You get personal expression in branch and bound method, too. We'll see where the personal – where the ability for you to put your stamp on this comes in.

By the way, that work will be simple. What happens now is we can't disagree. No one can do better than another person. If the rule is to solve the problem globally, to certify 100 percent accuracy, then you can't do that any better than I can do it because we're both doing it, and there's no lying and all that. We have to do that.

What can change is how long it takes, and indeed, your method of branching, or whatever, your bounding method could take four seconds, and mine might take two days to run, or something. That's how you would judge yourselves, is basically just how long it takes to run.

Okay, so the basic idea actually is very cool and very simple. Actually, the basic idea is quite useful in a lot of cases. In fact, I'll – we already assigned the last homework, didn't we? Hm. Do we have recourse? I would say so. We'll see. I'll think about it. I just thought it was good – there was a problem I've been thinking of that I wanted to assign. I even wrote some of it down somewhere. I'm gonna find out. So we'll see. That will be recourse in action. It'll be simple.

Okay, the basic idea is it relies on two sub routines that hopefully efficiently compute a lower and upper bound on the optimal value over a region. So that's what it is.

Now, how do you find an upper bound on a minimization problem over a region? That's easy. You could choose any point in the region and evaluate the objective function – I mean, if it's feasible. Well, if it's not feasible the upper pound is plus infinity, which is a valid upper bound. It's just not – well, it's not useful, but it's valid. You could run a local optimization method. It really doesn't matter.

The lower bound of the region you're gonna find by convex relaxation, duality, you could argue from Lipschitz or other bounds, or something like that. The way this whole thing's gonna go together is this. You're gonna partition the feasible set into convex sets, and then you're gonna find lower and upper bounds on each in the partition. From the partition you will then form a lower and upper bound on the global problem. And if they're close enough you quit; otherwise, you will partition and repeat.

Actually, before we go on, let me just write down the example, and I'll show you what it is. Maybe we won't put it on the homework. How about that? But I'll find this problem. Met with approval? I guess so.

All right. Let's just solve a problem like this. Let's minimize. Note the period. C transpose X subject to AX less than B. Then let's do something like this. Let's say that X one is either zero or one. Okay? So that's my problem. How do you solve that?

First of all, is it convex? No, not remotely. I guess the classical method is to – when you see that it's not convex, you back off slowly. Do not turn your back. You slowly back away, and then make a safe retreat. But, actually, how do you solve that? It's extremely easy to solve. How do you solve that problem?

Student: Plug in zero [inaudible] solve it.

Instructor (Stephen Boyd):Yeah, you'd plug in zero and you solve it, and you plug in one and solve it, and it's done.

Okay, so how about this one? How about that one? How do you solve that?

Student: Relax it, and hope [inaudible].

Instructor (Stephen Boyd):Yeah, that's – I mean, methods that don't involve prayer, or hope, or aspirations. But that would work. That would give you lower bound. What would you do for this one?

Student: Do all the combinations.

Instructor (Stephen Boyd):You could do all combinations if you had to. This is 1,000. You'd run 1,024, right? So you solve 1,000 – what would the answer be, by the way?

Student:[Inaudible].

Instructor (Stephen Boyd):What would it be? How do you get the answer? You'd run 1,024 LPs, and you just take the best one. Okay, so that's what you'd do. So branch and bound is gonna go something like this. This is sort of the basic – it's basically this idea, but doing it in an organized way where you're gonna scale it past ten. You can go to 15 this way, and so on.

By the way, just for fun – and then we'll get to the main thing next time – let me ask one more question about this. Suppose you had to solve that problem right there. You had to solve 1,024 LPs, but you had to do it fast. It was taking too long, your whatever – 15 line matlafting was taking too long. How would you – could you speed that up, solving 1,024 LPs like this?

Heck, let's talk about that. Could you?

Student:[Inaudible].

Instructor (Stephen Boyd):Oh, you did it in parallel. That's probably the easiest way to speed it up, is in indeed to find 1,023 friends.

Student:[Inaudible], and then kinda figure out which ones would be zero, and then kind of fix –

Instructor (Stephen Boyd):Oh, yeah. You could get euristic real fast by relaxing. You could also relax and get a bunch of them. By the way, if you relaxed, and some of these were slammed up against zero, and some against one, what can you conclude?

When you solve in relaxation, if they were all zero or one, what could you conclude?

Student: That's the optimal.

Instructor (Stephen Boyd):That's the optimal. If they were all zero or one, but three of them were just sitting in the middle, what could you conclude?

Student: Nothing.

Instructor (Stephen Boyd):That's the right answer: nothing. It is very likely that the ones that in the relaxation came out zero are indeed zero in the global solution, but it is absolutely not guaranteed. That would be one method.

Actually, in something like this, you could do a lot of stuff. Let's – we'll just mention a couple. First of all, you can do warm start. Whenever you have to solve a problem with the same structure over, and over, and over again, with nearly the same – not nearly – I guess it's not that near when you're taking the first five variables, and making them zero or one. But if there's 1,000 variables there you could use a warm start from the previous one.

Here would be another trick that would actually be quite useful. Suppose you've solved a bunch of these LPs. Your code is gonna have to have a logic in it that keeps track of the best one you've found so far, right? So you're gonna have – you're gonna maintain something like X best. And it remembers simply – it remembers some pattern of these, and then it solved it, and it's the best one. The others, by the way, are out. Anybody worse than X best is out of the running. Okay?

When you're actually now solving this problem using an interior point method, you get at each iteration a dual value. That's a lower bound. Everybody see? Right? You put logic in there that says the following: if you take a pattern like zero, zero, one, one, zero, whatever it is, something or other – you take that pattern, and you start solving it, the instant the dual value goes over this, what can you do? You can quit even though you haven't solved that one of your 1,024 LPs.

Why can you quit? Because by convex analysis, you have a proof or certificate that the optimal value of that LP exceeds this. But actually, that means, you don't even have to finish solving it to know that the answer will be worse than the best you've already seen. Therefore, you break. Everybody see this?

That will work really well because a bunch of these are gonna be way, way – it's gonna be obvious, right? It won't take the dual value very long to get above the best you've seen so far.

Student: Does that mean go above C transpose X best, or will it be –?

Instructor (Stephen Boyd):Absolutely, sure, because C transpose X best, that's the best you've found so far, right? Now you choose a different initial bit pattern. This was obtained with zero, one, one, one, zero – something like that. Now you're testing this pattern: one, zero, one, one – who knows. Like that. What happens is the optimal value of this thing may be like over – actually, if the optimal value of this is lower than this, then that’s the new winner. That’s the new X best. But if it’s above it, then the dual value will rise above this value, at which point you break.

Now, if it’s way above it – which is to say that this is a rather stupid, obviously stupid choice of initial bits here, then the dual value will go up, and be two iterations, and it’s all over. These are just practical matters, but they would allow you to do this a lot faster than it looks like it would have to be if you just did – if you didn’t do it, if you just wrote a five line thing that said, “For all 1,024 patters, evaluate the – solve the LP, and compare the optimal costs.”

So, I guess we’ll – how did that happen? Well, we’ll quit here, and we’ll continue this as our last topic next time.

[End of Audio]

Duration: 77 minutes

ConvexOptimizationII-Lecture18

Instructor (Stephen Boyd):Well, let me – you can turn off all amplification in here. So yeah, so it's still – you still have amplification on in here so you can – oh, well, we'll let them figure that out. Let's see, couple of announcements. It's actually kind of irritating, frankly. I wonder if we can just cut that off? Oh, the advertisement went away, that's good.

All right, I'll let them figure out how to turn off the amplification in a room that's got, like, a depth of 15 feet. So yes! No? All right. Anyway, I'll let them worry about it. Okay, first just a couple of announcements to – I keep getting hopefully. What do you think? Are we there? Getting better. It's funny. You'd think you could just turn it off.

But anyway, all right, okay. So a couple of announcements is, as you know, the final project reports are due tomorrow. So that's one deadline. The other is on Monday; I think it's been agreed. We're gonna have our poster session.

We haven't fixed the time exactly yet but we're thinking of something like 5:00-7:00 and we're gonna figure out about getting some food. So we'll – but we haven't announced that. That'll be on the website and all that kind of stuff when we figure everything out.

And there's already been some questions about what's the format of the posters and we don't know. So some people are gonna go ahead and do the fancy thing and make big posters which you're more than welcome to do but you don't have to. You can just print out 12 black and white slides, or whatever, and, like, you know, tack them to the poster board. And I don't remember what the size of the poster board is but maybe that'll also be forthcoming on the website or maybe be an announcement or something like that.

So the format, which is 12 slides, is – that's fixed and I've said it many times but, you know, please use our template. You're welcome not to, you don't have to, but then you better have better taste than we do. So, I mean, even – and there's certain things you just can't do because it's not acceptable and it's amateurish and stuff like that. And you all know what I'm talking about.

So, okay, so we're gonna finish up our absolutely last topic which is branch and bound methods. These are methods for global optimization. So in global optimization you have a non-convex problem but you're going to give up – you're not gonna give up a globalness. That's not a word, I just made it up.

But you're not gonna give up globalness – globality? You're gonna give up neither globalness nor globality but you are gonna give up speed. So what's gonna happen is these are gonna be methods that are slow but they don't lie. They will produce a – at any point you can stop them and exactly like a convex optimization method they will have a lower bound and they will have an upper bound.

And they're not approximate, they are correct and they terminate when the lower – when the upper bound minus the lower bound gets less than some tolerance and that's global. So this is a huge field. You can take entire classes on it. So this is just your taste of global optimization and it's basically just to give you the rough idea of how these things work.

There are other methods but they're all kind of related to this and then bits and pieces get very sophisticated. And even here I'll show you where there's lots of room for personal expression in these methods.

So, okay, so last time we started talking about this and the basic idea is this. It's gonna rely on two methods in your problem. You're gonna somehow – you're gonna parse up the region, you know what, there's actually amplification still on in here but it's cool. Actually it's not, but I mean, it can't be that hard to turn off the amplification.

Okay, so you're gonna have two methods which over a region of the feasible set given some description of a reason of a subset of the feasible set will compute a lower bound and an upper bound.

The lower bound, of course, is gonna be – that's gonna be the sophisticated one. Well, it can be sophisticated. That'll often be computed a Lagrange relaxation, by a duality, something like that. Oh, dear. Ah, the pleasures of – wow, now it's – I used to – well, a long time ago I did rock and roll sound so I know about this. These are bad monitors anyway.

So, okay, so the upper bound can also – can be done by a variety of methods ranging from extremely stupid and simple, for example, here's an upper bound. If your region is a rectangle you simply go to the middle of the center of the rectangle and you evaluate the objective there and the constraint functions. If that point is infeasible, you return the upper bound plus infinity, which is valid but not that useful. If it is feasible you turn the objective value at the center of the interval as the upper bound.

So that's, you know, they can't – it can't get any simpler than that. It's just evaluating the function at the center. You can do other things, too. Like, you can run a local optimization method starting from there you can do whatever you like to get a better point. Anything will work.

Matter of fact, these can have a huge effect on a real method and would be if you actually implemented some of this you would do this. We're not even asking you to do this on your last homework problem.

Okay, so this is the basic idea is you're gonna partition the feasible set into convex sets and then find lower and upper bounds for each. So that's the idea – and we'll quit. And we'll actually look at two specific examples of the general idea and once you've seen these two, which are a bit different from each other, I mean, actually they're kind of the same, you'll figure out how to do this in much more general – in the general case because it's really the ideas and not the details that matter here.

So here’s the way it’s gonna work. First we’ll do – we’ll just do simply unconstrained, non-convex minimization. Actually it’s silly; it could be constrained because you could build right into F the constraints by assigning F the value plus infinity outside the feasible set.

Okay, so we’re gonna optimize over this m-dimensional rectangle to some prescribed accuracy epsilon. It’s gonna work like this, we’re gonna have Q methods a – well, sorry, I’ll get to the Q methods in a minute but we’re gonna define phi min of Q as the global optimum of F, the global minimum of F over the rectangle.

So we’re just looking to compute F*, that’s what we’re actually gonna compute here. Okay, so we’re gonna have two upper and lower bound functions. So it’s gonna be a lower bound function and upper bound function. They must be above and below this minimum value and they have to be – one attribute they have to have is they have to be tight as the size of the rectangles shrink.

That basically even the stupidest method would satisfy this, so, the simplest lower bound, I mean, extremely bad would be the following. The simplest upper bound is the value of F at the middle of the rectangle and the simplest lower bound by far is to subtract from that number a Lipschitz constant, the diameter of the rectangle or the radius of the – diameter of the rectangle divided by 2 multiplied by a known Lipschitz constant on F. That does the trick.

Now that stupid one has the property that this – has this property that if a rectangle gets small enough then the gap between these two is less than your epsilon. By the way, you could use that as your safety backup lower bound, meaning that you can run a sophisticated lower bound based on convex optimization or something like that and then use this one if it’s better. And that means that it inherits the proof in that case. Such a method would inherit – such an algorithm would inherit the proof and so you would be safe from the proof police in that case.

So if there was a raid and someone said, “Can you absolutely prove your method works?” You’d point to a line of code and say, “Yes, it does.” So that’s really the only reason you would do that, by the way, would safety from them.

Okay, all right, so the idea is that these should be easy to – they should be cheap to compute because, of course, they could just be equal to this and that’s – but then they’re impossible to compute. So, in fact, the tradeoff here is gonna be something like this. You want cheaply computable – you want bounds that are cheap to compute but good.

Good means merely that they are close enough to each other in the instances you’ll look at that branch and bound will run quickly. So that’s sort of the key to everything here is getting cheaply computable bounds.

Okay, so let's look at the branch and bound algorithm, it works like this, basic one in this case goes like this. You – I call my lower bound method on the initial rectangle and I call my upper bound method and I call these U1 and L1.

Obviously I can terminate if U1 minus L1 is less than epsilon because then I'm done and I'm absolutely done. By the way, this method can also return a point that achieves this upper bound. So – if you wanted to say, "Okay, I'm terminating and I want to return an X then it will be the job of this thing to return the bound."

By the way, it's the job of the lower bound to return a certificate proving it. So if you want to make the algorithm sort of really formal and correct and all that when it quits it should quit returning two things, a certificate that the upper bound – that this value – that this upper bound is actually truly and upper bound.

Best certificate being to demonstrate a point which has that objective value and which is feasible and a lower bound. We're gonna find out what the lower bound looks like in a minute. It's much more sophisticated than merely a dual feasible point which is what you have in convex optimization.

Okay, so that's – we'll see what this is in a minute. Okay, now if this does not hold – so in other words, your upper and lower bounds are not less than your tolerance what you're gonna do is you're gonna split the rectangle into two subrectangles.

Okay? So – and you're gonna do that by choosing a variable to split on. So a rectangle, of course, is something that looks between – it's the rectangle is described by two vectors, an L and a U, a lower and an upper bound. You choose a component to split on and then in fact in more fancy versions you could decide how you want to split it.

Do you want to split it evenly or not evenly and this is where all the personal expression goes into these methods. So if you had some clever method where there was any reason you didn't want to split it evenly that would be fine.

But you'll partition it like this and then you'll call the lower bound and upper bound methods on these children. And, in fact, this is – let me just draw a picture here to show how this works in R2. I mean, it's silly in R2 but here's the point.

So here's your initial rectangle like this and you evaluate the upper and lower bound and if they're not within tolerance you split it. And you could either split it along these variable and you're gonna partition two rectangles, you could split it along X1 or along X2 like this. And let's say you decide to split it along X1 you can further decide how you want to split it. But here it doesn't say, it doesn't specify so there's a splitting like that.

Now what you do is you calculate the lower bound here and the upper bound here and let me just do a – I'm gonna fill in some numbers just so we can see how this looks. And, in fact, if I do this example you'll kind of understand absolutely everything about this. So let's start with this one. I call the upper bound and I get ten, I call the lower bound and I get six.

So that's the – so we know that the optimal value of the function in here is somewhere between six and ten. That's all we know, you don't know anything else. What can you say about the function value like at this point here? What is it?

Student:[Inaudible].

Instructor (Stephen Boyd):It's bigger than – well, what do you mean it's bigger than six and ten?

Student: It's [inaudible].

Instructor (Stephen Boyd):Ah-ha, okay. I thought you were gonna say that. It was a trap. I wanted you to say that. It's false. Half of it is true. It's bigger than six because the minimum I can provide that this six means I can prove that on this rectangle the function never gets smaller than six. Why? Because I have a lower bound on the function of six. Okay?

What's the – so I can tell you if I pick a point here the function value here is bigger than size. I cannot say it's less than ten. What can I say though? I can say this, there is a point in here, maybe not that one, but there is a point in here which has a function value of less than or equal to ten, that I can say.

By the way, I'm skipping, you know, I'm assuming the [inaudible] and the [inaudible] are achieved in all that. You know, I mean, it's true anyway, the F is – let's make F continuous or whatever. It doesn't matter.

So there is a point and, in fact, it would be the job of the upper bound routine to return that point in case it were challenged. Actually, in case our – suppose our epsilon were five in which case we're done. We've calculated the global optimum within five it would be the job of – so there is a point, let's call it there.

But this doesn't really matter. Okay? So, all right. So now we're gonna partition – we're gonna go like this and I'm gonna call my lower bound function here and here and my upper bound function. And let's actually – if I go through a couple of – I'll just ask you some questions and I think then you'll know everything there is.

And let me switch these around so the lower bound comes first. Okay. So here let's see what would happen – well, let me ask you a couple questions. Let's call the lower bound function and let's suppose it were five here. Any comments? That's the lower bound. Is it a valid lower bound?

Student: It's a valid lower bound.

Instructor (Stephen Boyd):Yes, it’s a valid lower bound but –

Student: It’s of no use.

Instructor (Stephen Boyd):- it’s of no use. And, in fact, if I got five there what could I do with the five? Immediately replace it with –

Student: Six.

Instructor (Stephen Boyd):- six. Okay. So the point is we don’t require – we do not require that the lower bound and upper bound function should return better answers than their parents. But, in fact, we can just quietly copy – in other words, calculating this lower bound was a complete waste of time if we get five. Everybody got that?

So if you get this – I could just replace it with six because basically – and if someone challenged you and says, “How do you know the function is bigger than six in this whole region?” You say, “Look, I know it’s bigger than six in a bigger region. Therefore it’s bigger than 6 in here.” Okay, good. So that’s one discussion.

Okay, so let’s suppose it’s seven here and eight is the upper bound. Okay? So any comments? By the way, what if this were 11? Be very careful. So what’s that?

Student:[Inaudible].

Instructor (Stephen Boyd):It could happen. You can’t say anything stupid has happened yet but I can now. Right? Let’s look at that. What can you say now? Does it bother you? How much does it bother you? Is it impossible or merely stupid? But which one is it? Are these valid? Could these be valid bounds? Yeah, sure. Okay.

So you can insist on the following, when you call the upper bound method on the children at least one of them is as good as the upper bound in the parent. Everybody got that? The reason is this, the – in fact, when you call the upper bound on the parent you could ask for which – for the location of the point that achieved that number then that child – whichever child contains that point can’t have an upper bound that’s worse.

So this can’t be. One of these two can be modified to be 10. Everybody follow? Okay, by the way, it’s just – once you understand all this you understand everything. I mean, but you actually really have to think about these things because they’re not obvious.

So, I mean, well, they are obvious. Sorry. It’s just they’re confusing but you have to draw this picture and then think about it very carefully.

All right, let’s do one more. How’s that? It’s fine. Okay? And now let me ask you, suppose this is what the two children – so you’ve taken the two children, you’ve called the lower and upper bound method on both children and they’ve returned these bounds.

Now I want to know what's the global lower – what's the new lower bound? What can you say if someone tells you – asks you, you just – let's see, in the first iteration you called both of these methods once each, right? Now you've done it three times. So you've got three times the work invested now and the question is, "What is the new lower bound – what do you know how that you didn't know before?"

Student: It's between seven and nine.

Instructor (Stephen Boyd):It's between seven and nine.

Student:[Inaudible].

Instructor (Stephen Boyd):Okay. Hang on, let's just go very slow. What's the new lower bound?

Student: Seven.

Instructor (Stephen Boyd):Seven. Okay. And the argument – and that's better than it used to be which was six. So originally it was six, meaning that whatever else happened we knew that the minimum had to be bigger than or equal to six. Now we know it's bigger than seven which is the minimum of these two numbers. Right?

Because in this region we trust these methods. It says that the function value is never smaller than seven here, it's never smaller than eight here. By the way, do you know which side the minimum is gonna be in?

No, not yet. We're getting there, okay? No, you don't know. You still don't know. But your new global lower bound is seven. What's your new global upper bound? Well, it can't be – well, it could be 12, it's valid. I started by saying, look, the optimum value is absolutely less than or equal to ten. Now what can I say?

Student:[Inaudible].

Instructor (Stephen Boyd):Ten?

Student:[Inaudible].

Instructor (Stephen Boyd):Yeah, no, we've agreed on it could be either side. You know there's still amplification on? Well, it's a good way to end a year of –

Student:[Inaudible].

Instructor (Stephen Boyd):What? What? What? What? It's good now. Okay, great, great. Let's just – okay. Yeah, so what's the new upper bound?

Student: Nine.

Instructor (Stephen Boyd):Nine. Explain it to me.

Student: There's a point [inaudible].

Instructor (Stephen Boyd):Good. Okay. So if I told you, "No, no, no, no, no, the global low – you know, no, no, no, no, F* the minimum on the whole thing is 9.5." Then this thing's lying. So there's a point on the left whose value is nine and therefore the global – so we went from six nine after this iteration to seven nine. So that was our – that's literally the – so the interval of ignorance was this and it went down to this.

Everybody got this? Okay. So, I mean, these are not complicated you just have to go to a quiet place figure out the mins and the maxes and all the logic. It's elementary but as you can see you actually have to think about it for a minute to figure out what it all means.

And let me just do one more thing just for fun. Let's do this. That's 9.5, okay? Now what can you say now?

Student:[Inaudible].

Instructor (Stephen Boyd):What can you say?

Student:[Inaudible].

Instructor (Stephen Boyd):The optimal point is on the left side, okay. And what are the new bounds?

Student:[Inaudible].

Instructor (Stephen Boyd):Seven and nine. So which is the same we had a moment ago except we've actually now made an additional statement which is that the optimal solution must or can lie in the left-hand side. Must, right? Okay, and the – what's the argument? What's your argument? How do you argue it?

Student: That's the point of the left side that has a value lower than the lower bound on the right.

Instructor (Stephen Boyd):Yeah. So the point is that there is a point in here which has a value of nine. Every point over here has a value that exceeds 9.5 and therefore this whole thing goes away and we're back down to – and now we've actually learned something at least in terms of the location of it.

And so this is called pruning. We're gonna get to all of this but if you understand all these ideas of which is about five or six, they're all quite elementary and all that, you understand branch and bound. That's – basically that picture, that's all of it.

So but let's go over it now. Okay, so here's what happens. Okay, so you're gonna split it, you're gonna update the lower bound so the lower bound is gonna be the following. It's going to be the minimum of the lower bound; the new global lower bound is the minimum of the lower bound's returned by the children.

And the upper bound is also the minimum of the upper bound returned by the children. And by the way if you wanted to also put in the U1 and L1 you could do that as well here. The new – because you can't – you could query the children, get two answers back and effect you could have no progress whatsoever towards improving things. Right? None.

It's nothing in the semantics of the lower and upper bound methods that requires you to make progress. And so I won't go back to that example and show how that works.

Okay, you find the partition and repeat steps three and four and let me just ask a couple of – well, we'll get to that. Okay, so that's how this works. Okay, now as I said you can assume without loss of generality that the upper bound is not increasing an LI is non – is also non-increasing. Okay? So that's right.

So – have I got that right? So as you go down the lower – the upper bound is gonna go down. By the way it has the option of staying level if you don't get anything better. And LI is – I want to say that LI is non-decreasing, don't I? I do. That's a typo. LI is non-decreasing. Okay.

So what happens is when you repeat this several times, what you do is you have a binary tree and so let me just show you over here what that's gonna look like in this case. So let's go back to this and let's suppose now I choose to split this guy and I split it like so and then I split this one and I get that – something like that. Then what I'm really developing in this case is a tree, a binary tree. And so it would look something like that.

Let me draw this, I'll have to make it a very fat tree. So you get something that looks like that and this node corresponds to this original rectangle. This node corresponds to this left rectangle, right? And I could even label this cut as L and R for left and right, okay?

Then this node corresponds to this rectangle which I then split and that happens to be top and bottom. So I can call this Top and Bottom, for example. And then these nodes – this node corresponds do this rectangle and the bottom corresponds to this rectangle. This one was further split like that and then this would be L and R, okay.

So this partially developed binary tree corresponds to this partition here and let me just make a couple of comments about it. So the leaves of this tree correspond exactly to this set of rectangles, okay. So that's what the leaves are. By the way, the non-leaf nodes correspond to rectangles also in here but ones that are further split, right? So that's what this thing looks like.

Okay, on every node here I've calculated a lower – every node here I've called the lower and the upper bound method. So I have a lower bound in all of these and if you have a lower bound – if you look at the leaves, right? The leaves constitute a rectangle partition of the original set.

In other words, it's a set of rectangles whose union gives you the original rectangle and whose intersection has – intersection of any pair has non-empty interior. That's a rectangular partition. Did I say non-empty? I meant to say empty interior, okay. So that's a rectangular partition here. So you have this thing.

Okay, now if the lower bounds on these things were, for example, you know, 5 – 5.5, you know, 4 and 8 what can you give? What is the lower bound for the entire thing? What's the lower bound on the whole function? I just drew the lower bounds on those nodes.

Student:[Inaudible].

Instructor (Stephen Boyd):What is it? What?

Student: Eight.

Instructor (Stephen Boyd):That's the lower – I want the lower bound on the optimal value of the parent – the top rectangle.

Student:[Inaudible].

Instructor (Stephen Boyd):What's that?

Student: Four.

Instructor (Stephen Boyd):It's four. Do you know where it is? Well, not until I draw some upper bound there. If I draw upper bound you'll know where it is. You might, you might not. Every upper bound there could be 20 and that just means it's weird but, I mean, it's perfectly valid. Your lower bound is four, okay. And now we can actually describe – I can tell you exactly what a certificate of the lower bound looks like in a non-convex problem.

If I send a halt message to this process and say, "Stop, I'm done." Let's see, you just did one, two, three, four, five, six, seven, you just did – did an effort of solving seven. Let's say each lower bound is a convex problem you've solved. So you just solved seven convex problems, say, "That's it. I'm out of time. I quit."

It returns a lower bound of four here. And then you say – then you go back to the method and you say, "Prove it." Right? So how do you prove it? So in a convex problem you prove it by producing a dual certificate. How do you prove the lower – what data – what object do you hand back that proves it?

Student:[Inaudible].

Instructor (Stephen Boyd):No, there's no – this is abstract, there's no Lipschitz constant, you don't know anything. I mean, it's an abstract lower bound method.

Student:[Inaudible].

Instructor (Stephen Boyd):There's no point – if there a point that achieves it you're done. If you have a lower bound – if you know the global optimum is bigger than or equal to four and you have a point whose value is four you're done.

You don't have a point. The – you might not even have a feasible point yet. All the upper bounds everywhere could be labeled plus infinity. So how would you prove it? If someone said – if you said, "I happen to know that the function value in this big rectangle is bigger than four." Someone would say, "Prove it." How would you do it?

Student:[Inaudible].

Instructor (Stephen Boyd):Okay, but so? You have to return a – you can say, "Here my proof. My proof is this that in this region every point in that region has a function value bigger than five. Every single one in this one has a function value bigger than 5.5. Everything here has a function value bigger than four and everything here has a function value bigger than eight."

So what you return is you actually return the partition with a lower bound on each one. Everybody – so that's your certificate of proof. So, in fact, when you finish a branch and bound method what's – instead of giving the actual certificate proving your lower bound is actually gonna be a partition of the original space with each element in the partition will have its own lower bound.

And if you want to certify that lower bound you have to go to whoever provided that lower bound and it could be Lipschitz, could be duality, Lagrange. I mean, who knows what it is, right? It could be anything you like, any lower bound method you know.

But that would be the matter of the lower bound method. That particular lower bound method; it could be different lower bounds by the way for each one, right? Yeah?

Student: Based on the lower bound with the non-leaf nodes?

Instructor (Stephen Boyd):Oh, no, I have a lower bound here, too. Yeah, definitely have a lower bound there. Here, I'll draw it. It's 4.5 here, something like that. There. Yeah, let's put some lower – and when I did the whole thing I started with three and then this could be also 4.5.

Student: You have the lower bound to be 4.5.

Instructor (Stephen Boyd):In this case? Uh-oh, you're right, sorry. I'm doing it the wrong way. Thank you. Sorry. Yeah, I shouldn't make it four. I meant to the other way around. I was working on it. How about 3.8 and you're gonna have to help me on this – three – how about that? That look reasonable?

Yeah, but the way I drew it, it would have been 4.5 and that's because we're – despite whatever this slide says in the top line we're actually assuming that the LI's are increasing because that's always a possibility, right? So if you call a lower bound method on a subrectangle and you get a worse lower bound than the parent you just take the parents lower bound. So, okay? Everybody got this? I mean, these are – it's very, you know, it's simple and all that but you actually – it can easily get confused. Make sense?

Student:[Inaudible].

Instructor (Stephen Boyd):Where? Over here? It's quite likely. Well, what is it? You don't like it? Oh, you mean this?

Student: Okay, about [inaudible].

Instructor (Stephen Boyd):No, no, that's okay. That's cool.

Student:[Inaudible].

Instructor (Stephen Boyd):Yeah. That's fine. It means basically that the function value – so what's the problem with it?

Student:[Inaudible].

Instructor (Stephen Boyd):There probably is one but – tell me what you don't like about it. You should all be checking this, by the way, this reflects on you. In fact, it reflects on you more than me because I don't really care.

But imagine what other people will think if they watch this video and they think, "You gotta watch this." It's like, "There's this whole class from Stanford they – this guy said something so stupid it's amazing and they just sat there happily." So it'll reflect on you more than me. I think it's cool. What do you think? Everybody cool with it? All right. Yeah?

Student: What is the lower bound [inaudible] is, like, like solving the same kind of problem but there's no [inaudible]?

Instructor (Stephen Boyd):No, no. It's totally abstract. This actually – heterogeneous methods for lower bounds could be being run for each node and totally different. One could be based on a Lipschitz constant; one could be based on Lagrange duality. You could fire up four lower bounding methods on one, you know, for one region and return to the caller the best of them. One could be like an SDP bound; one could be a SDP relaxation, one a Lagrange dual. I mean, it makes no difference.

So, no. And then you have the option if you get a bunch of lower bounds and you'd pick the best one. You can also pick the best one of the calling parent because that's also a stupid lower bound, right? Because if you look at a subregion it can't have a lower bound – well, it could be worse, but it's just silly than the [inaudible]. So, okay?

So did we reach stationarity? I think we're cool. Okay, so – okay. All right. So let's see – so this is the basic idea. In fact, all we did was we walked through – whoops. This, this business here. And then this would go – this actually now would go recursively through the tree. And so in fact it's the same as saying it's the minimum of all of the leaf nodes. Okay?

So to – by the way to make this algorithm – to full specify it you need a couple of rules. You need to know things like this, which rectangle do you split? So you have this binary tree – well, you have a partition and you can split a rectangle so that's – you have to choose which one do you – so for example, on the next step someone says, "Okay, I give you one more step. You have to split like this guy, you know, one of the leaves here you're gonna split, so one of the four.

Then when you decide to split a rectangle you'd have to decide to split it like, you know, along which variable? In the 2D case you want to split it either vertically or horizontally but in the M dimensional case which I guess is this, you choose which of the M variables you're gonna split on. Okay?

And, you know, it's immediately obvious you can do things like if you had eight processors split could mean you could not divide it in half but divide it into eight sections and send – and then send the 8 subrectangles or whatever to your eight processors or whatever. So, you know, these are all kind of obvious things.

Okay, so here are some rules – this is where the personal expression – this is one of the areas where personal expression comes in is in choosing these things. One is you split rectangle with the smallest lower bound along the longest edge and in half. I mean, that's kind of just a default method.

But in any particular application maybe you can think of a better way and you can experiment with some and find that one will work really, really well and stuff like that.

And you can mumble all sorts of idea about justifying this but they are absolutely nothing, you're just mumbling. I mean, there's no reason to prefer one of these – I mean, you can talk about that there's probably stupid methods but this – there's lots of other methods, other splitting rules and things like that that can be justified just as well as this one.

Student: So in every example we split one of the leaves?

Instructor (Stephen Boyd):Yeah.

Student: And we'd get new upper and lower bounds on that leaf and probably get them all the way up to the root or no?

Instructor (Stephen Boyd):Yeah. So let's see what happens when you split. Let's choose one and split it. I mean, assuming, you know, that we're not in – I'm already in some big trouble here or whatever.

So in this case if we follow this rule we would take this four here and I would split this four. And so that's gonna be this guy, I think, right? And so I split it like that and then it would – I would develop my tree this way and now my two diagrams have hit each other. So I would get this – this was 4 and now you're gonna have to help me make some consistent things here.

So let's see what would happen. Let's make it interesting and let's imagine progress is made in lower bound and I would do that by saying this is 4.5 and I have to be very careful, is that right?

Am I doing it right? Yeah. And then this one could be 4.2. Okay? And if that were my two new lower bounds what's the new global lower bound? 4.2. So we just made progress in the lower bound. Okay?

So that's how that works. So that's – so the idea – the rough justification, I mean, this is like a depth-first search or something like that. So the rough justification of this is you're going after – pulling the lower bound up as soon as possible. Okay, so that's it. Okay, so here's an example but we've already done an example so – yeah?

Student: What happens if they return the same lower bound?

Instructor (Stephen Boyd):No problem, it's fine. You want me to do it? Here, let's go over here and do that. Yeah, so the lower bound of what? Like, you want 4.0 and 4.0? Like that? You tell me, what just happened? Tell me, what happened? That can happen.

Student:[Inaudible] the lower bound [inaudible].

Instructor (Stephen Boyd):Well, it's not worse.

Student: No, it's not.

Instructor (Stephen Boyd):It means you just called those two methods – how much did you learn?

Student: Nothing.

Instructor (Stephen Boyd):Nothing. At least looking at the lower bounds. In terms of lower bound you made no progress. It's fine, no problem. That's perfectly fine. It's nothing wrong with that. So that's – yeah, that's fine. That's just not a problem. Okay, let's see. Oh, what if the lower bound had been five on one of those? What would that mean?

Student: Just [inaudible]?

Instructor (Stephen Boyd):What's that?

Student: If one of the smaller [inaudible] of five just go back and replace it with 4 again.

Instructor (Stephen Boyd):Well, you could but you could also just label that leaf. I mean, you can now cross off that node – that rectangle, and say – can you? No, you can't.

Student:[Inaudible].

Instructor (Stephen Boyd):See?

Student: It depends on –

Instructor (Stephen Boyd):I fooled myself. Okay, all right. All right, scratch that. This would be a good time to turn on some feedback in here, by the way. I could blame it on that. Okay, no don't, please.

Okay, so this is the picture. Okay, so we've already talked about this – pruning goes like this. It says that if there's a rectangle whose lower bound exceeds the upper bound – the known upper bound across the whole thing then it can be marked – it can be pruned.

And that means you can sort of put an X – well, it means lots of things. It means actually everything you can now prove you know for sure that the solution is not in that point. And that – sorry, in that rectangle. Okay?

And that can actually happen while you're running and that would allow you to free memory, for example. Actually it would have – if you're using – if you're splitting on the lowest lower bound it – you'll never split that rectangle anyway because it's got a high lower bound. So you'll never split it. You'll be splitting others.

So, in fact, it doesn't change the algorithm in the slightest. What it does do is allows you to free memory. So what would happen is if all of a sudden a point were discovered with value – uh-oh, now I have to be super careful here.

If I – suppose the – these are all lower bounds, I haven't talked about upper bound. Let's suppose that in this set I found a point with upper bound 4.2. Now let's be super careful. By the way what's our global lower and upper bound at that point?

I mean, it’s at least as good as 4, 4.2. Right? Because I have a point with value 4.2, it’s in – left, right, that’s top – it’s in this set, it’s in this rectangle but – and I know that across the whole thing the bound is 4.0, okay. Now what can I – now let’s work out what I can conclude. You do this, too. I did this as the examples, not prepared and these things are tricky. So help me. Tell me, what can I conclude?

Student: You [inaudible].

Instructor (Stephen Boyd):I can get rid of what?

Student: The rectangle.

Instructor (Stephen Boyd):I can get rid of which one?

Student: The left one.

Instructor (Stephen Boyd):This one here?

Student: Yeah.

Instructor (Stephen Boyd):Absolutely. So it is now known absolutely certainly that the global solution does not lie in here. So this one is pruned officially.

Student:[Inaudible].

Instructor (Stephen Boyd):This one is pruned. That’s gone. This one is now officially gone. And so that means, let’s see, you’re not here, that’s top, you’re definitely not here, that’s - is it this one?

Student: Right.

Instructor (Stephen Boyd):Like that. And that says that at this point still in the running are these two, okay. Everybody got this? Now actually, by the way, you can even go up a tree farther and prune something below that. So, by the way, this is not gonna change the iterations, especially if you’re going after the lowest lower bound first.

It’s not - it won’t - because you don’t choose things with high lower bounds. What it will allow to do is call free on a bunch of these and just free the memory. It also, by the way, if somebody stops and says, “What’s the progress on the algorithm?” You could say, “Well, I had,” let’s see, “One, two, three, four, five - I had five active rectangles two of which,” or something like that. I don’t know. “Before the algorithm I had five or something and now I have only two, so it went down.”

And you could also talk about the total volume. In this case the total area. So you could actually say – well, actually you can say something now. You can say the global solution is in this little – it’s in this rectangle here, period. Everybody see?

Student: And we just like to destroy your certificate or is there another way?

Instructor (Stephen Boyd):Have I what? Destroyed what certificate?

Student: Of –

Instructor (Stephen Boyd):Oh, if I free those?

Student: Yeah.

Instructor (Stephen Boyd):No, I store them. If this is – if it's like lawyers around and I'm gonna need it in the end to make my certificate I store a copy of them, yeah. But, by the way, had I developed this node I can safely destroy those records, right? So I can safely destroy the records that went below this but I have to save that one in case a lawyer asks me to prove when I terminate that the lower bound is bigger than 4. Right?

So I keep those records but I can destroy everything below them. So - okay. So this is pruning. And, like I said, it doesn't affect the algorithm but it can reduce the storage requirements or whatever. And then you can track the progress had been total pruned volume or unpruned, and the number of pruned leaves in the partition. So – and that gives you a rough idea of how complex the problem is, right?

Because when I terminate and say, you know, "Okay, I'm done. The lower bound is 4.11, the upper bound is 4.18," and you stop and someone says, "Really? Prove the lower bound." If I say, "Oh, yeah, no problem. By the way, it's a convex problem. I give you one dual point and I'm done." Right? Or something like that.

If it's a non-convex problem we go, "Oh, yeah, sure. Let me tell you how I know the lower bound is bigger than 4.18." You say, "Here is a partition of the original rectangle into 40,000, you know, 40,346 rectangles. And each one, by the way, I've attached a list – a lower bound for each one and the minimum of all those numbers is 4.18."

And so that's how you – that would be the certificate. Now that actually tells you sort of how complex your function is essentially and you can actually work that out. I mean, if it's something with a bunch of lower bounds, I mean, sort of minimum or something like that it would have to look something like that.

Okay, so let's do the convergence analysis quickly. So if you don't prune every time you split a rectangle you take a child – you take a leaf, sorry, you take a leaf, and then you append two children to it. So it's no longer a leaf but what had been one leaf is now two so the number of rectangles after K steps is K - the number leaves which is the number of rectangles in a partition is K.

And the total volume of all the rectangles and certainly the volume of the initial rectangles and that says that the minimum volume of all your rectangles is less than or equal to the initial volume divided by K.

It’s actually probably a lot smaller but that’s a lower bound. And that would only occur if you split these, you know, I mean, you could get a much, much better lower bound if you’re actually splitting things in half so the volume’s going down by a factor of two you could probably – it’d be a log in there or something like that. But it doesn’t matter. I mean that’s the – this is good enough.

And what that says – because this is gonna be a really, I mean, crappy proof. It’s just gonna basically say it works and not much more because actually there’s no point investing and having a fancy proof that gets a better complexity of result but they’re all gonna be terrible. They’re all gonna be exponential in the end. So it doesn’t – it really hardly matters then who has the best exponential bound.

So this says the following. After you’ve done a large number of iterations there’s at least one rectangle with small volume. Now we’re gonna show that small volume implies small size.

Now, of course, in general that’s completely false, right? If you keep splitting a rectangle – just in R2, if you keep splitting a rectangle, you know, vertically then it’ll have small volume eventually and its diameter will still be the same.

So you have to – we’ll see that you’re gonna have to – it has to be something in your selection rule which controls for that. Now, if you have a rectangle that has small diameter then by definition it – sorry, by our assumption it says that our – the bounds there are small, okay.

Now that’s interesting because when you picked that rectangle L was the leading candidate, was the lower bound. And if that was the global lower bound when you picked it and you got a U that’s close to it, that means that your global upper bound is very close to your global upper bound and that means that UK-LK is small. So this is gonna – that’s how that proof is gonna go. And these all require little bits and pieces of the assumptions there. We’ll see.

Okay, so to prove that the – to insure that the – if you want insure that small volume implies small size then you need to control essentially the condition number. And the condition number is – for a rectangle it’s easy, it’s just the longest edge divided by the shortest edge.

And what you need is the following, if your splitting rule is you split along the longest edge then you’re absolutely guaranteed that the condition number of the new rectangle is less than the condition number of the previous one in two.

Now there’s many other rules that would satisfy this and it really hardly matters because these are not – in some ways these algorithm are in the worst case they’re extremely slow. So it’s not interesting that they converge, of course they converge, I mean, unless you do something stupid, they converge.

And you generally can't prove that they converge faster than exponentially so it's – I'm not exactly sure, I mean, this is just enumerating some of the intensely stupid things you could do to make this not work.

But, okay. So – and let me just explain this because it's not totally obvious but it's actually true. It's this – if I have a rectangle, let's take a rectangle like that, what's the condition number on that? Just roughly? What's the condition number?

Student: Four.

Instructor (Stephen Boyd):Four, fine. And if I split along the longest edge like this what's the condition – the maximum condition number of the children?

Student: Two.

Instructor (Stephen Boyd):Two. So I made progress. My condition number went down. Everybody – okay. Now the question is, can you split a rectangle and have the condition number go up?

Student: Yes.

Instructor (Stephen Boyd):Yeah, like what?

Student:I could split this.

Instructor (Stephen Boyd):Split what?

Student: What you had horizontally.

Instructor (Stephen Boyd):No, no, I'm sorry, yes, sorry, I meant to say this. Can you split a rectangle along its longest edge and have the condition number go up? Could you?

Student:A square?

Instructor (Stephen Boyd):Yeah, let's take a square. What's the condition number on that? One. Let's split it only – and let's say that this edge is 1.0001. Okay? So that's the longest. I split. What's the condition number on the children?

Student: Two.

Instructor (Stephen Boyd):Two. Okay. So the condition number can go up when you split along the longest edge. However, it can't go more than two. So you'd have to argue that this is the worst thing that can happen and that's not too hard to do. That is true. Okay?

So it says when you split a rectangle along its longest edge the following is true. The condition number is less than or equal to the maximum of the condition number of the rectangle you split, two. And the two would be achieved if the condition number of the parent had been one, which is to say it was a cube.

Everything cool on that? I mean, I didn't show it but, you know, it's like arithmetic. So that's what that is, okay. So that says if you use the longest edge splitting rule the condition number will never be worse than the initial condition number and the max of that in two, so period.

There's lots of others, it doesn't – the only thing you can't do is you can't keep splitting in the same direction or something like that so that the volume goes to zero and the longest – so every now and then you have to split against the longest edge or something like that. I mean, it doesn't matter.

If you bound the condition number you can bound – then you're done because you can bound the diameter. So you would have this, I mean, the volume is this thing and that's bigger than or equal to I can assume the worst thing would be – the smallest it could possibly be would be to have one of these things have the max and all the others be at the min.

That's this thing and so you can bound this by condition number is hardly surprising. And so this tells you that if the condition number is bounded, which it is if we use the longest edge splitting rule, then it says that the size of Q is small.

And so that actually finishes it. That's done. So that's the whole proof now and there's a few things I didn't show. Oh, I know it's not – I didn't formally show that when you split a rectangle its condition number is no worse than the parent or two, whichever is the worst of those two. So that's all. So that's it. Okay, so that is – that's branch and bound.

We're gonna look at one more example of it, just a different flavor, exact same ideas. This will be more specific about it. So it's gonna be mixed Boolean convex problems. These come up all the time. So it's a problem that looks like this. You minimize a function and we're actually gonna assume that these functions are all convex in both X and Z.

Now a little bit weird here because these functions don't actually have to be defined for the Z's except Boolean Z's, right? So and in fact they – if they're not defined for those you have to extend the functions FI to make sense when the Z's are in between, okay.

And there's actually many ways to do that but that has to be done. So okay, the X is called the continuous variable and the Z is called the Boolean variable. By the way, it could be a problem with just Boolean variables.

By the way, if there's no Boolean variables that's called a convex problem and we can solve it very effectively. And what happens is this, if you commit to the Boolean variables here, so in other words if I've got ten of them and I commit to them, I take one of the 1024 patterns of Booleans this problem becomes convex in the X's and therefore it's easy to solve.

So, by the way, if N here is ten then you can always solve this problem by just taking all 1024 variations on pattern – bit patterns on Z, solving those convex problems. Right?

Okay, so that's the brute force method is you simply evaluate – you simply solve this problem for all two to the N values of Z. And if Z is less than ten or these are small problems and you have a lot of friends with idle machines and, you know, you know how to distributes jobs on different machines or something, you know, that can work fine, I mean, provided N is, like, small, like less than 20 or something like that. That'll work fine.

But – well, we'll see what that corresponds to. By the way, the partition of the feasible set is actually gonna be a discrete partition. It depends on the values of Z. Okay, branch and bound – if you run branch and bound on this what's gonna happen is in the worst case you're gonna end enumerating stuff anyway.

So it's – that's – there's no – and it's not hard to create problems where that's exactly what'll happen except with more overhead than just writing a bunch of for loops that goes through and tries every value. Because you maintain some big tree and the tree just fills up. You make a full binary tree and then you are exactly the same as if you had – you just filled it initially or something like that. But the idea is that it'll work much better and it generally does.

So in this case we can talk very specifically about how to find a convex relaxation. The simplest one by far is the linear relaxation. So you take Z, which had been in the set squiggly, you know, left bracket zero, one and now you make it in the interval, square bracket zero, one. That's the interval here.

Now, you know, if anyone – and people ask you what you're doing you can make up all sorts of cool stuff. You could say, "This is the quantum mechanical interpretation. You see the Z's before had been true or false now they have some – they're some number between zero and one." And if they buy it, you know, great.

But, anyway – or in communications you would call this a soft decision, right? And you'd say, "No, no, pardon me but I'd like to know whether that bit is a zero or one." And you could say, "I'm working on it. At the moment it's a .8." That's a soft decision or – actually, do they – they have these in these [inaudible], right? People have seen that, no, somewhere?

So that's – so it's got all sorts of names in different application areas where they make it sound like it's more sophisticated and more useful than actually the zero, one's. But anyway that's a good excuse.

Those are good techniques to learn. I mean how to relax a problem and then convince the people you're working with that although you're no longer solving the original problem you're solving something that's more sophisticated and actually has more information in it, you know?

Like in fault detection, they could say, "I just want to know if the fault has occurred or not." And you go, "Ha, that's a very unsophisticated way to look at it." I'm saying it's occurred, you know, .8 and they go, "Well, what does that mean?" And you go, "Well, it's more sophisticated. I mean, I – it means it's probably occurred." Or anyway – so we won't go into – I won't go into that. Let's go on.

Okay, now obviously when you solve this problem which is convex you get a lower bound on P*, that's obvious. And you can get plus infinity, that's actually a very, very nice lower bound because that means that the original problem is infeasible. Now, by the way, how would you get an upper bound? How would you get an upper bound for a problem like this?

Student:[Inaudible] the combinations, right?

Instructor (Stephen Boyd):Yeah, yeah. You can test any, right? By the way, how would you get it though? Tell me some practical methods for this thing. Well, here we can look at this several ways but we can – I'll go over some of them but here's one. The simplest method by far would be to round each relaxed Boolean variable to zero or one, okay, which, by the way, can destroy feasibility at which point your upper bound is plus infinity, okay.

Another option is you could round them and once you've rounded them go back and solve for X's. That can only make things better. You could, for example, restore feasibility, reduce the objective value, okay. That's your other option.

You could use a randomized method, I mean, this is – you would generate a random ZI and 01 with probability equal to this relaxed value. So now the .8, the soft decision, makes sense. So you generate random ones and try this and so on. Let's see, I could say – let me say one more thing probably more useful in practice than a randomized method which, however, will impress people. So I'd recommend it on that end for that reason.

But there's actually a much better way to do this. What you can do is you can do a local optimization. So once – assuming you have a feasible Boolean point you simple cycle through the Z's, you flip each bit. So if Z17 is one you flip it to zero and then – and see if actually things got better. If it got better you keep it.

So, I mean, you're just talking about you do some really greedy algorithm for this, can only improve your upper bound, can't make it worse. So, by the way, methods like that often work – I mean, just basically in general convex relaxations followed by the stupidest local optimization methods you can think of, the stupidest, greediest methods work shocking, just shocking, well. So that's actually way to get a good upper bound.

Okay, how do you branch? That's easy. You pick an index K to branch on and you do the following, you pick index K and you split your problems into two, one where ZK is zero and one where ZK is one, okay.

And you relax – how do you get the lower bound and upper bound's on the thing? The lower bound you relax these, you get a lower bound here; you relax these you get a lower bound here. Upper bound you can do whatever – however you like. You can round the relaxation followed by a local optimization, that's gonna work very well.

Okay, so these are just Boolean convex problem within -1 Boolean variables because you can eliminate the chosen variable if you like and you get exactly the same thing as before. So you can just run branch and bound. How do you get the new lower bound? I mean, this is the same as before so I'm not even gonna go into it because it's kind of obvious. You get these new lower bounds.

So here's the branch and bound algorithm for a mixed convex Boolean problem. You do the following, you just form this binary tree, you're splitting, relaxing and calculating bounds on the subproblems. So each – now each node is labeled – when you split the binary tree, each – the edges are labeled by a certain Boolean variable either equal to zero or one. That doesn't say the left and the right sides in the tree.

And so actually a node is very – a node in the tree is very interesting in that case. So in that case this is the wrong picture and in this case it would look something like this. There, so there's a tree and it basically says that you first split on variable 17 and so this would be Z17 equals zero, this is Z17 equals one. This is Z13 equals zero. Z13 equals one and so on, okay.

So now the meaning of a node is quite – very interesting. So it basically means – it says here if I go to this node here it means that two out of M of my Boolean variables have been fixed. Two of them, and they've been fixed to the values 1 and 1 because I took this thing.

Here they're fixed to the values one and zero here and then below this. But the others are still floating. I haven't committed to them yet so that's the picture. And if you develop the full Boolean tree then congratulations you just did brute force on it. So, okay. So there is no convergence proof because it's stupid at most when you've filled out two to the N leaves you just did brute force. So it's silly.

Obviously you can prune. Same thing, you can pick a node with a smallest L and to pick a variable split that's actually if you read about things – again, this is where, you know, personal expression comes in, so one of the places. The other is in the – when you actually implement the methods that produce the lower the upper bound. That's where the – honestly that's probably where the more important personal expression comes in.

But this is also important and there's two, actually, you'd find defended in the literature. One would be the least ambivalent. So you'd take the ones where you – or 01, the idea is to prune that quickly, to just prove that the solution is not there. And so that's one method.

Another one is maybe the most ambivalent and here's what you do is you choose the value of K for which ZK* minus half is minimum. If, by the way, you get to – whoa, what does it mean if you get to a node and the lower and upper bound are equal? Actually, what if you get to a node and at that node when you do the – solve the relaxation it's a 01 solution in the relaxation?

It means the lower and upper bound are equal and it means you now know the global solution on that node. In other words, if those binary variables leading from the top down to your node have those values you now know the optimal value below it, okay. So but the most ambivalent would be to pick the one for which that's minimum, I mean, that's one method.

Okay, here's a small example just to see how this works. I think you're gonna do exactly this by hand on the homework. So it's the only way to really clarify your mind about these things and that way you can say you've done branch and bound. You can even have a nice pause then so the person gets it and then you can lean forward and say, "By hand."

So which very few people can say that actually. So it's a cool thing to be able to say and then if they looked shocked you can say, "I mean, I didn't solve the QP's by hand but I did do branch and bound by hand." But anyway, okay.

So here's an example, we start off – and let's just even back out what on earth this means. It's a three variable Boolean LP, this is stupid, I could have solved 8 LP's and be done with it. Okay? So I could have. But let's see what this means. Let's just see what these numbers mean to see if they make any sense. I solved the original relaxation and I found out that the – I relax all three Boolean variables I find out the lower bound is -.143. What's the infinity mean?

Student: Infeasibility.

Instructor (Stephen Boyd):What was infeasible?

Student:[Inaudible] that one was one the problem.

Instructor (Stephen Boyd):No, no, this infinity. What does that mean? It has a meaning. Oh, it means – yeah, it means infeasible but what's the precise meaning of that infinity? It has a very precise meaning. I mean, it sounds informal but it's completely precise.

Student:[Inaudible] infinity means nothing.

Instructor (Stephen Boyd):I means that whatever upper bound method I used – rounding, whatever, couldn't have been that great actually but whatever upper bound method I used it did not produce a feasible point. In which case the upper bound remains plus infinity.

By the way, it couldn't have been a very good upper bound because there's only 8 – but it doesn't matter. So the point is it could have been just I rounded the relaxed solution, evaluated, violated constraints which are N plus infinity. I split on Z1 and I got here's 0 and 1. I got – and by the way, what does this mean? That means something, too.

Student:[Inaudible].

Instructor (Stephen Boyd):It means this says that when I solve the relaxation with Z1 equals one, by the way, the relaxation there is I'm relaxing Z2 and Z3. There's just three variables here, it's kind of a stupid problem, right?

So I relaxed those two and that LP was infeasible and you know what that tells you? That tells you that if Z1 is zero there's no feasible point. And so we will never – basically you know what that says? That tells us at that point we – at this moment we know exactly what Z1* is. Well, it's still – we still don't know, by the way, the problems even feasible. So I should say Z1* is either nan or 1 and that's gonna – we're only gonna find that out later.

But at the moment it cannot be zero. It's either nan or one, okay. So here's z1 equals one and I solved this LP, the relaxed one, and I find the lower bound is .2 which means that I've actually gained information on the lower bound but I still have yet to produce a feasible point. You know, who knows why? Because I have a very bad upper bounding mechanism or something like that.

Then I split on Z2 is zero and one, this can't be the answer and then here I get 11 and now I'm done. And let's figure out what I – how much work did I save myself? Actually, how much work did I save myself by running branch and bound?

Well, let's see, I could have just solved eight LP's and been done with it but instead I solved one, two, three, four, five, okay. So, I don't know, I cut it by a factor of two. I mean, it's a stupid example, right, but that's not a significant one, but okay.

We'll see significant ones. Now we're gonna see a significant one. So we're gonna go back, we're gonna solve the same problem we were solving in the L1 lectures. There we were doing heuristically. We want to minimize the cardinality of X, subject in a – we want to find the sparsest vector in a polyhedron.

And so you write that as a mixed Boolean LP, you have lower and upper bound on the Z's and I'll get to those in a minute. Those you get simply by bounding the box and I mean, putting a box – the smallest box around the polyhedron, okay.

So the relaxed problems are simple enough. If you exactly relax this it comes out to this. This, by the way, we saw. It's the sophisticated version of L1 heuristic for minimum cardinality when the box is asymmetric with respect to the origin or something. By the way, if any of the ranges – if L1 is positive or U1 is negative the sign – sorry, whether or not that variable is zero or not is done, okay. So that's fixed.

Okay, so this is – we're gonna run – we're gonna split the node with the lowerest bound. Oh, there is one thing you can do. When you're solving this minimum cardinality problem the outcome of value is clearly an integer. So if I tell you that the – if I solve that LP or whatever it was and I get a lower bound that is 19.05. What is the actual? I can actual immediately say that the lower bound on the cardinality is what?

Student:[Inaudible].

Instructor (Stephen Boyd):Twenty, it's 20 because the cardinality, which is obviously an integer, it's bigger than 19.05 and the next integer is 20. So this is obvious.

Okay, so we'll do a small problem. This one has got 30 variables and 100 constraints so – but the number is, you know, these are not small numbers, right? Two to the 30 is about a billion or something. And it turns out it takes eight iterations to find a point with global minimum cardinality. That's the good news.

Now the bad news – and this is extremely typical of branch and bound. It took a lot more iterations to prove that the minimum cardinality was 19. And by the way, these numbers get even more crazy when you put in a sophisticated local search because then basically it's very typical, you find it in six iterations or something like that, you know, very fast and then 80,000 iterations later you've now proved that the point you found on the sixth iteration is actually like 5 percent suboptimal or whatever your thing is.

Okay, so this took 309 LP solves, that's including the 60 to calculate the lower and upper bounds. And so these were actually just done by Jacob and actually all the Python code is on – you do not want to write this in matlab. That would be a joke. Well, not joke it wouldn't be funny at all for you. It would be funny for other people but not for you.

So here's what sort of the tree looks like at various points. This is kind of what it looks like. And by the way, when it finally returns and says it's 19 and someone says, you know, "How –" or whatever it is, it wasn't 19, whatever the minimum cardinality is 30 something. Is it 19? The minimum cardinality is 19.

It basically says, "Look, you know, how can you prove this?" You would say something like this – you'd have to return this thing. Actually, you'll see what you would have to return the certificate proving it.

This just shows the global upper and lower bound. The dash one shows the true upper bound. You can see it starts at 12 and ends up here at 18 in 60 steps or something. And then you can't see it but it terminates right there when this thing goes up and you're done.

So that's sort of the picture and you might, by the way, this kind of puts it all in perspective. It basically says that these L1 heuristics are awfully good, right? Because you've produced a point with a cardinality of 21 and the global minimum was 19. By the way, had you done an iterated L1 or a local search here you almost certainly would have got a point with cardinality 20 and you would have got it just in a handful of LP solves. So – but it took a whole lot more effort to prove it, so this is very common.

This shows sort of the portion of non-pruned values. Out of the 2 to the 30, this shows you the fraction. And, I mean, it's not – it's just something to kind of look at and see how progress goes. And the number of active leaves in the tree; I guess it terminates at 50 something or other. So what that says if that if someone says, "Prove that the optimum – the minimum cardinality is 19." Then the way you'd do it is you'd say, "Not a problem. Here is an X with 19 non-0's that satisfies the N equality."

And they go, "Fine. That means it's no more than 19. How do you know there's not a point with 18 non-zeros?" And you'd say, "No problem," and you would return this data structure which is a tree with 50 some odd, 53 different – a tree with 53 nodes which is a partition of this space and on each one of those you would have solved this convex problem, you'd have left the evidence or the certificate for that there and you'd say, "There you go. There's your proof."

So your certificate is a complicated object in this case, okay. This is a larger example, I'm just gonna zoom through it because we're – well, we're out of time. But – and this is just in the interest of intellectual honesty just to show that if you try to solve larger problems – by the way, these are problems with just not many variables, right? Convex problems with 50 variables are joke. As you know, you can solve a convex problem with 50 variables and a hundred constraints, what are the units?

Student:[Inaudible].

Instructor (Stephen Boyd):Okay, milliseconds. I'll accept that, at the way small end of milliseconds. I would – I'm going for microseconds on that one, but no problem. By the way, thank you for saying that on the last day of the year. That makes me very happy.

So but the point is here, you know, the minute you go to the non-convex problems even like 50 variables, 100 variables, these are now very – these are large numbers. And you can be lucky, of course, and solve gigantic problems again if the gods of global optimization branch and bound are smiling on you, you can actually solve very large problems.

But you better understand it's only by luck or something like that, that that happens. So I think I won't go into this. I mean, there's a huge problem and this problem is great because it kind of – at least it sets the record honestly.

So this was run 5000 iterations and at that point here's what you know about the global solution. It is between, you know, something like 135 and 180 or something like that.

And this is really the way these – this is kind of how it – when it fails, this is what it fails like. This is the way it looks.

Notice how much better your point was with all that effort. So and this picture would be a very good endorsement of these L1 methods as heuristics. This is why it's better off saying, "No, I'm vague about scarcity. I don't really need the absolute sparsest." In which case you're way better off right down here.

So we are out of time. You're gonna hear another communication from us about the exact – we'll set the time and we'll probably give you the poster format for Monday and well, maybe we'll see you tomorrow and then maybe – definitely Monday.

So oh, and I have one more thing I have to say. I have to thank the TA's very much for creating all the new lectures. One very important thing. Someone not here but who has, I mean, helped enormously in the last two classes is Michael Grant. So some of you know this by saying something doesn't work and having the build number increment because of you, literally within an hour.

So we all owe a huge debt of gratitude to Michael Grant who – I'm not even gonna tell – he'll never even find out. Somebody will point him to this part of the lecture then he'll hear about it.

Okay, well, thanks and we'll see you Monday.

[End of Audio]

Duration: 81 minutes