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