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