ConvexOptimizationII-Lecture11

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Student:[Inaudible] for the row then?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Student:L1.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Student:[Inaudible].

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

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

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

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

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

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

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

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

Student:I did.

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

Student: No. Instructor:

Why not?

Student: Same excuse came here last fall.

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

Student: Yes.

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

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

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

Student:[Inaudible].

Instructor (Stephen Boyd):Yes.

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

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

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

Student:[Inaudible].

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

[End of audio]

Duration: 78 minutes