MITOCW | Lecture 9 | MIT 6.832 Underactuated Robotics, Spring 2009

RUSS TEDRAKE: OK, welcome back. So last time, I tried to ease us into a switch in thinking. From instead of trying to explicitly solve the Hamilton-Jacobi equation, I wanted to try thinking about a different class of algorithms which we called policy search. Where you explicitly parameterize your control system with some parameters and then just search in the space of parameters for a good solution.

So the idea was, let's go ahead and define some class of control systems by some parameter vector. And then, our optimal control problem, which we called sort of minimizing J pi of x0 t. We could think of as explicitly minimizing over that vector alpha. Something that if I shorthand pi of alpha with just alpha of x0 t.

And if you're willing to restrict your thinking to a single initial condition, then you can really just think of it as, I've got some old function J of alpha, and I want to minimize it with respect to alpha. That puts you squarely in the land of you can call fmin. In MATLAB, you can do sort of any old optimization on it.

So today, I said the lecture is called Trajectory Optimization. I tried to point out that thinking about this policy search, that could be a general thing. That could encompass feedback policies. For instance, this could be parameterized by if K is filled out with alphas, that's OK.

And I also said it could be open-loop trajectories, right? In general, I could just ignore x, and it could just be-- let me see how I wrote it last time. It could just be, let's say, that u at time t is just alpha of n where n equals some floor of to over dt. I'm not sure that's the best way to write it, but that's, I think, a clean way to write it.

So what does that mean? That means that my control policy over time is just a set of-- it's a zero-order hold trajectory. Where each of these are dt long. And this one is alpha 1, this one is alpha 2, and so on.

So if I'm willing to make some sort of simple tape of trajectories that are parameterized by these alphas-- and naturally, you can do a cleaner job of doing this with splines or whatever. But let's think about the simple representation. Then solving this min alpha J alpha is equivalent to trying to find the open-loop trajectory that I'd like to follow which minimizes J, for instance, for some initial, for a particular initial condition.

So that class, this sort of the open-loop family of control policies is special enough that there's a lot of methods that are highly tuned for that open-loop trajectory optimization. So I want to talk about a few of them today. They're very powerful.

They tend to scale to fairly high-dimensional systems. And I actually think they can be used as a part of a process to design good feedback controllers. But that's a longer story. Let me just tell you today how to solve open-loop trajectories.

In the trajectory optimization world, there is roughly-- well, there's lots of ideas. And so many ideas, actually, that it's going to slip into Tuesday I decided. But let me tell you about the first two of them today. I want to talk about first shooting methods and then direct collocation methods. And we have lots of half-baked examples that were coded in the middle of the night, so to hopefully bring the message through.

OK, so what's a shooting method? You might be able to guess. Shooting methods are-- how many people know? How many people know what shooting methods are? Excellent, OK.

How would you characterize a shooting method? What would you--

AUDIENCE: Can we integrate the states from the book from the beginning time and the [? co ?] states from the end time and hope they match up in the middle or something?

RUSS TEDRAKE: Perfect, well, yeah, I mean, in general, actually so it's even simpler than that. In general, so shooting methods are often the title for solving boundary value problems, which is what you just said. The name comes from boundary value problems.

We can use them more generally, even if it's just an initial value problem. But the basic idea is exactly what you said. Let's just simulate the system with the parameters we have and then work to change our parameters, shoot at a little bit different place.

So let's say we have t over x for some very simple system. And I run my controller from initial conditions here, and I get some trajectory with alpha equals 1, let's say, or even alpha equals some long vector. Maybe it's 1, 2, 43, 6. I get that.

And let's say my goal is to get my final conditions to be here. Then, what I'm going to do is, I'm going to change alpha and run it again, shooting successively until I get to my desired final value. If I change alpha, then maybe my controller gets me here. And if I change it again, maybe it'll get me all the way up to the goal. I see you.

AUDIENCE: What's the update?

RUSS TEDRAKE: Yeah, OK, I'll tell you the update. But the big idea is, I'm going to I'm going to try to solve a problem, for instance, a boundary value problem, by starting with some initial conditions simulating and just changing the parameters. So you can imagine, if the thing you're trying to solve is not-- I mean, a boundary value problem is obviously one thing we can do. But maybe you also have a cost that you're trying to optimize over that. Then the basic idea still holds.

So I told you the first way to start thinking about how to do that last time. We can evaluate J of alpha pretty easily. Let me stick with my superscript alpha notation. We can evaluate that with just forward simulation, right?

And if you want to know how to change alpha, then it helps to know the gradients-- partial J, partial alpha, evaluated at the current alpha. I told you one way to do that last time, with an adjoint method. Which I still can't resist calling back prop through time, because that's-- I learned it first as a neural networks guy.

And there's a second way to do it, which in some cases, is not less efficient. But it's certainly easier to derive and easier to code. So I want to do that to make sure we have the intuition about it too. It's also useful.

Again, this is the neural network name for it. There's probably a good name from more standard optimization theory. But in neural networks, people call it real-time recurrent learning, which is RTRL. And this is BPTT.

So how do we compute J of alpha? The adjoint method, I told you that if you simulate the system forward in time, you get the sort of forward equation. You figure out J. If you then simulate the adjoint equation backwards in time, I called it y. y dot is some function going backwards.

You can interpret y as being the sensitivity of changing-- so let me just write down the form of it quickly. So the adjoint, remember, was x dot equals f of x going forward. Did I actually right down the right equation for you going backwards?

And then, negative y dot is f of x y minus G of x T. Where those are the big gradient matrices I wrote down last time, going backwards. And then, the update gives you, that partial J partial alpha is just a simple integral now.

So this backwards equation, which happens to be the adjoint equation we saw in Pontryagin's minimum principle. y has an interpretation as the sensitivity of the cost on changing x-- y at some time T, y at time 3. It's the same size, the same dimension as x-- this y variable.

It's a column vector just like x. It has the interpretation, it's the sensitivity of J on changing x at that time. So you compute forward, then you compute back the gradient of J with respect to x of t. And then knowing that, you can simply compute the full gradient with respect to the alpha.

And that took a little bit of derivation through the Lagrange multipliers. RTRL is actually even simpler. Is it OK if that wall disappears for a minute? We wrote that last time, right? I'm just going to go to my third wall here.

This time, I'm only going to simulate forward in time, which is why it's called real-time recurrent learning. Because some people don't like having to simulate forward to capital T and then simulate all the way back to make an update. That's sort of, we have to go all the way to the end of time in order to make your update at time 2. It's more appealing if you can make your update at time 2 by just thinking about time 0 to 2. So you can do it in a forward pass only.

The name came from maybe that, I think, at one point in time, someone maybe thought maybe this is what the brain is doing. Because people thought the brain can't be doing these backward passes efficiently. So maybe real-time recurring learning is what the brain is doing. But I don't think people really think that anymore. There was one paper that thought that, which is a nice paper, but--

So let's do, let me just show you RTRL, because it turns out to be maybe even the more simple thing. So I have J of alpha starting from x0, 0 is the integral from T. Oops, I did that again.

Let me come up with a new working variable. The previous one y was useful coming backwards. If we're going to go forward, let me define a matrix, P, where the ij-th element is a partial xi, partial alpha i.

If I have that and I want to take gradients with respect to this, then I can do it pretty easily actually. If I want to do partial J partial alpha, I can go inside the integral. It's just dt. I'm just using my chain rule here. I get partial G, partial x, partial x, partial alpha. These are x at some time T, plus partial G partial U at time T, partial pi, partial alpha.

And in general, if I have a feedback term in my policy, I also got to worry about partial G, partial U, partial pi, partial x, partial x, partial alpha. It's just a chain rule derivative of this. Do you agree?

It turns out, using the matrices we had used before, I can write that very simply as integral from 0 to T of Gx, the big derivative of x times P, this matrix here, plus the big derivative of G on alpha. If we know partial x of t partial alpha, then it's easy.

So how do we get partial x T partial alpha? Well, that's easy too. Let's look at the forward equation here, xu which is, in this case, pi alpha x of T.

So let's look at how this changes with alpha. Let's take the derivative with respect to alpha. I'll write it even more cleanly-- partial f, partial x plus partial f, partial U, partial pi, partial x times P, which is partial x, partial alpha. Let me write it as--

Everybody agree with that chain rule? No? OK, ask me. Yeah.

AUDIENCE: So there you have G with respect to x and then P [INAUDIBLE] And then we have G of alpha which would be the second variable. What happened in the third one?

RUSS TEDRAKE: G of x is actually partial G partial x plus partial U-- G partial U, partial pi, partial x.

AUDIENCE: Oh, I see.

RUSS TEDRAKE: That's the big gradient with respect to x.

AUDIENCE: And that I on the bottom is J. Is that correct? [INAUDIBLE]

RUSS TEDRAKE: This is now the matrix P.

AUDIENCE: [INAUDIBLE]

RUSS TEDRAKE: Oh, thank you. That should be a J on the bottom. Good, thank you. Please do catch me on those things. Yeah, thank you. So then, are we happy with this?

So this is pretty simple too. Just now, I get an equation forward in the gradients P dot is my big f of x. Which I get is the direct and the indirect gradient with respect to x times P plus the gradient of F with respect to alpha.

And that's it. I'm almost waiting for more to do, right? So if you're willing to go-- if you want to go forward in time, then if you're willing to keep around this extra term, partial x partial alpha, then as you move forward in time, you can build up your gradient of partial J partial alpha. And by the time you get to the end of time, you don't have to go backwards again. You know what the total derivative is.

So why would you use this versus the other method? Does everybody agree with that? I don't see big smiles, but this is satisfying and simple-- smile.

The cost of this is carrying around this matrix. So potentially, that could be big. If you have a lot of parameters, let's say I have 100 parameters in my control system, or 10,000 parameters in my control system, then you're actually integrating forward a matrix equation that could be pretty big. It's something that's x-- the dimension of x by the number of parameters.

So that's one-- that's really the only problem with it. The back prop through time is only carrying around this y, which is the size of x. But to do this sort of nice forward-backward update, you'd better be able to remember the trajectory that x has taken over time.

So for very long trajectories, it might not be much more efficient to do this. It's just a trade-off. Some problems are actually quite nicely done in RTRL. Some are nicely done with back prop through time.

The back prop through time is, the reason it's so beautiful and clean is that-- so remembering your goal here is to compute a vector, partial J, which J is a scalar with respect to a vector of parameters. Here you have to carry around a matrix to do that, partial x forward.

The back prop through time takes advantage of the fact that at the end of time, everything collapses to a scalar again. And that's why it only has to sort of carry backwards. If you willing to go all the way to the end of time, you can remember only the effect of that scalar value, j, with respect to the x's. So that's why you're allowed to carry around less of this. But it involves going forward and backwards.

Is that intuitive? Yeah? What can I say more? There's various other reasons why you might prefer one or the other.

So for instance, let's say you have a final-- a boundary condition. Let's say you want to solve this constraint, minimize this function subject to the constraint that x of capital T is my goal. You can write that constraint into either of them. I find it more natural to write it into this RTRL form. Just because you know explicitly partial x, partial alpha.

So if you want to compute at the end your constraint derivatives, saying what should I have done? How should I have changed alpha in order to enforce that constraint? Then it's actually a simple function of P.

So maybe I'm not giving you a silver bullet by telling you one way or the other. But depending on exactly the way you want to use them, sometimes one is more useful than the other. So in the simulations I'm about to show you, I'm going to actually RTRL, just because it's also, this is trivial to code.

But the big idea here is really just that I'm doing a shooting method. I'm simulating this thing forward. And I'm trying to compute what's the change. And that's my scalar cost with respect to a change in my parameters. So that I can then update my parameters.

And last time, we talked about multiple ways to do that. You might do that with a simple gradient descent algorithm. You could do gradient descent. You could say that alpha at the n plus 1 step is just alpha n minus some learning rate, eta, times partial J partial alpha.

And I argued last time that you can do better than that by sequential quadratic programming. To the point where SQP methods are so robust that you just find something like SNOPT, you download it, and you use it. And I'll try to convince you as we run some of these things that apart from installing a package, which takes a few minutes, once you do it, you'll never go back to this.

I'll tell you one reason right off the bat. Choosing the learning rate is a pain in the butt. You never have to do it again. So?

Let's see some examples. Let me do the pendulum with SNOPT. So using SNOPT is just, all you do is you tell it-- you give it a function which can compute J and J, the derivatives of J. It's SNOPT stood for sparse nonlinear optimal-- or optimization.

And in a lot of problems, many elements of this vector, partial J partial alpha, are 0. We're going to see that in our direct collocation methods. A lot of those gradients are 0. It happens in the one I just told you. This is typically not-- typically all of those elements of that vector are non-zero, which means I'm not actually getting an advantage by using the sparse.

NPSOL is the non-sparse version. But I don't think it's much worse, if any worse. Just for those of you that are trying to figure out which package you want to convince your PI to buy. I think SNOPT is normally pretty good. And there's a student version that we can have you download that'll do small problems.

So let's do the pendulum. My cost function here, is just an LQR cost. I have Q as a-- I have J as the integral from 0 to capital T of x minus the x goal. I had to be careful about wrapping. But transpose Q x minus x goal plus u transpose Ru. The whole thing, dt, right?

And I actually also have a final value cost where Q is 10, 1. Qf is 100 times that or something. Q and R is 100.

So it's going to reward the thing for getting to the top. Let's see what it does. I'm going to plot the trajectory for every step of the optimization. So it's not quite doing simple gradient descent.

It's now going to do a sequential quadratic programming update where it estimates the quadratic bowl and tries to jump right to the minimum of the bowl. So it's going to be a little bit more jumpy, as you see it, but it's fast.

It's finding the last optimization and then boop, right to the top.

AUDIENCE: I have a question. So how is this trajectory parameterized? Is it just openly.

RUSS TEDRAKE: It's exactly this, the floor of T over dt. My sloppy notation was because it's MATLAB notation.

AUDIENCE: So when they say [INAUDIBLE].

RUSS TEDRAKE: That's the open-loop policy, which had U at time T, was just, I do a floor, and I go to the n-th index in alpha. So alpha 1 is my control action from 0 to dt.

AUDIENCE: So how many parameters is this case? [INAUDIBLE]

RUSS TEDRAKE: Good. So I did two seconds covered by 40 bins. I bet if I do 20 bins, it's OK. It may be a little less faithful to the-- less smooth. But it works out.

So please realize that like 60% of that time was just drawing those plots to the screen, easily, right? Especially when it's reshaping the screen and stuff like that. That's wasted time. So hey, we could do the same thing on the [? karpal. ?]

AUDIENCE: Can we do this without [INAUDIBLE]?

RUSS TEDRAKE: Good question. Yes. So yes, it's quite simple. In direct collocation, it's really simple. So actually, my code does it for direct collocation and doesn't do it for the gradient. But it's actually quite fine.

You just have to have T as one of your parameters tucked into alpha and be able to compute partial J, partial T, which is not very hard actually. You just have to figure out how this function changes when you take a derivative with respect to T. And it's just, it's like an x dot times the quantity J at T. It's not too bad.

If you can take that gradient, you can optimize with respect to it. So what about for the [? card pull? ?] Well, oh I forgot, I took off the zooming. So I gave it a fixed axis.

It's going to make a liar out of me. This is the slowest I've seen. There it is. Let me do that again. It's certainly more impressive than that.

It's starting from random initial tapes, by the way. And every time I've run it for the [? card pull, ?] it comes up with the same solution. Come on. There we go. All right, not quite as impressive as it was in the lab, but that's still pretty good. If I turn off drawing, it bet it's a lot faster.

So I turned off drawing, there you go. I think my computer is slower on battery power too. That's probably-- I'm disappointed. Oh well, it's still pretty fast.

AUDIENCE: So I was just wondering, if you simulated longer, would it stay at the top, or would it fall in?

RUSS TEDRAKE: So I actually put a final value constraint in on that. So it actually gets to 0, 0. So because I'm simulating, it probably would stay up, right? But I think the natural thing to do would be to draw an LQR controller in at the top for instance.

And in fact, we're going to talk on Tuesday about how to LQR stabilize that whole trajectory. Because for the most part, I think open-loop is just the first piece of the-- just is one of the tools. Good.

So what happens if we did-- you can do the acrobot too. I'll do the acrobot in a second here. But I also have the sort of simple gradient [? descent ?] version in here. That if I just did my alpha equals negative eta times dJ d alpha in there, let's see how that does.

Was that faster? Let's do it. Let me try that again. That's not running. I didn't save it. I knew that was too good to be true. OK, my ears are too big. I probably have to change my learning rate. Thereby confirming my complaint that--

OK, so it works. But never do it again, you don't need to. Just download SNOPT. It'll get there eventually. You can see the errors going down. And if I set my learning rate properly, it'll go down pretty fast.

But not fast enough for me to be patient. And then, here's the SNOPT version again. Now the pendulum is very fast. Good.

Let me do direct collocation before we get too mystified by the simulations. There's another idea. So shooting methods are certainly subject to local minima. And I've got an example that I'll show you in a few minutes that I hope will make that clear why they can be subject to local minima.

Something that people do, they do sometimes multiple shooting methods. Sometimes it's sort of, there's even numerical sensitivity sometimes, sort of integrating this thing for such a long time. So a lot of times, people will define some breakpoint in the middle of there, some artificial breakpoint in the middle of their trajectory. And say, I'm going to optimize, first try to get me to this point.

And then I'll say if I simulated from some other point to here and then use as a constraint in their optimization to try to make that residual go to 0, if that makes sense. I'm just not going to talk about multiple shooting. But just know that there's a version that people use often, which are the multiple shooting methods.

To some extent, direct collocation is maybe the extreme of that. I told you there's a lot of good reasons to use SNOPT, or some SQP. So first of all, why use SNOPT?

No learning rate tweaking, that's a big one for me. It's often faster convergence. Because you are doing big steps. You can sometimes jump over small local minima.

But there's a big one in there that's not on the list yet. What's the big one I'm going to say? What's perhaps the best reason, I think, to use SNOPT?

AUDIENCE: Fewer constraints?

RUSS TEDRAKE: Good. It's easy to add constraints. Because the way these sequential programs are solved, there with these interior point methods and things like this. And they're very efficient at handling constraints. So in my pendulum swing up on the simple gradient descent, I didn't actually have a final constraint on getting to the top in the SNOPT version. It's just trivial to add that.

I can put bounds on my actions very easily. I could say, do gradient descent, but never let U at time T be bigger than 5. I can even put constraints on the trajectory if I wanted to.

So because of the power of nonlinear optimization to handle constraints, people came up with a different way to hand the optimal control problem to an SQP method that exploits those constraint-solving abilities a little bit more explicitly. And that's the direct collocation methods.

AUDIENCE: Can I ask a couple of questions? When we were talking about this previous case that you showed, isn't it just providing the [INAUDIBLE] function, providing the Q's and R's is sufficient to actually get [INAUDIBLE]. So if we add an R constraint on top of it, it would be [INAUDIBLE] on top and sort of like a 2 with respect to our method? Because the goal we have essentially is to maximize or minimize the [INAUDIBLE] over the trajectory. We can put some content, like it would be more information. Like for example, I wanted to reach this state or that state for sure. We [INAUDIBLE].

RUSS TEDRAKE: So I think that's a very RL way to think about it-- not cheating. I mean, cheat, cheating is good. If you can hand more information to your algorithm, do it. Don't worry about cheating. But no, I agree. So the question was, is it fair to give it a final value constraint, or am I comparing apples and oranges, roughly, right? Like if I say one is not using the final value constraint and the other one is.

So in my opinion, the goal is actually to get there at time T. The optimal control program I'd like to solve is minimize some, even minimizing just u transpose Ru dt subject to x of t is my goal. That might be my favorite way to write it down.

And then, just the question is, what methods can I use to solve that? So the opposite view of the world here maybe is that because a lot of the methods don't handle these constraints explicitly, I'm stuck writing down a cost function, which is x transpose Qx plus u transpose Ru, even if that's not explicitly what I want.

Or maybe I should say, especially the closest analogy is if I have a final value cost, straight and maybe I make Qf really big. The only question is what you really want to do. In most cases, I really want to do that. So I'm quite happy to use solvers which could do either case. I think a more powerful solver is one that can handle either case.

AUDIENCE: The other question is, if we want to solve something which takes a lot of time, like T is relatively big, it seems that if this open-loop policy thing that you're following has one parameter per time step.

RUSS TEDRAKE: It could have a lot of parameters. Good.

AUDIENCE: But is it possible to just describe the whole policy with a very limited space with very few parameters and solve for that?

RUSS TEDRAKE: So I tried to be careful to write down the equations. So the question was-- can people hear the question or not? The question was roughly that if we're worried about a problem with a very long horizon time, it seems that I might have to have a very large list of parameters to cover-- to make a tape that's that long.

And so, aren't the algorithms rather inefficient there? Couldn't I do better by writing down maybe a feedback policy? That's often the case is that feedback policies can be more compact.

So I tried to write down all these equations as if there was some dependence on x in your policy too. So the equations will be the same if you do that version. The only thing that I don't handle nicely in the things I'm throwing up on the board is that I'm always simulating from the same x0. So I'm really explicitly optimizing, even if I optimize a feedback controller from a single initial condition, its performance from a single initial condition.

So you can quite easily say make a different cost function, which is, let's say I want J to be the sum of my performance from initial conditions K through 100. And this would be, let's say, J of xk, 0, or something like this. Maybe I could start it from 100 of my favorite initial conditions. And that would try to optimize a feedback policy perhaps better.

But the only thing that's not nicely addressed, I think, is choosing your initial conditions in a nice way. DP-- [INAUDIBLE] program handles that beautifully. And these things are much more local methods. So they have to be in there.

But I absolutely agree. Oftentimes, the open-loop tapes are not a particularly sparse way to parameterize a policy. Good.

So the direct collocation methods, like I said, more explicitly-- they're even more in the sense of open-loop policies versus feedback policies, actually. But they also more explicitly add these constraints.

So here's the idea. Let's make my alpha my vector that I'm trying to optimize over. A list of, can I call it u0, u1, u2-- is that another reasonable way to describe what I've already done-- to u capital N. I've got a list of control actions.

But now I'm going to actually also, I'm going to augment my parameter vector with the state vector. So I'm just going to make an even bigger parameter vector. One of the reasons to do that is, I can now evaluate J of alpha, which is this 0 to T, G of x. Maybe I even approximated this discretization, right? So it could have a dt in there or not, it doesn't matter to me.

If I have u and x and all these things directly in my parameter vector, I don't actually need to simulate in order to evaluate that. I can just evaluate it immediately. I have x and I have u.

The only problem is that how do I pick alpha so that x and u are consistent? It better be the case that x1 looks like the integration of x0 with u0 applied had better get me to x1. So instead of having that sort of implicit in my equations, let's make it an explicit constraint.

So let's do this subject to the constraint that x of N plus 1-- actually, lots of constraints, so it's a list of constraints. x1 had better be f of x0 u0 times dt, let's say. It had better be equal to 0, and so on, then x2-- right?

So if I'm willing to add representation here, I can actually evaluate my cost function without explicitly simulating. That's cool. I can take gradients very quickly, because now it's just explicitly the gradients, partial G, partial x. Well, I know x, right?

AUDIENCE: Wouldn't you also want x1 minus f of x0 u0 dt minus x0? [INAUDIBLE]

RUSS TEDRAKE: Yes, thank you. Thank you. So plus-- thank you. Good, thank you. I had some weird mix of discrete time and continuous time floating around there.

So if you parameterize it like this, you can very efficiently calculate the gradients. For instance, whereas you can easily calculate the gradient with respect to time, in this parameterization, and you just add a lot of constraints to your solver. And you're asking your solver to solve for a lot more points.

It turns out that these solvers handle constraints very efficiently. In fact, it's often times more efficient to add constraints to the system, because it reduces the search space. So this is actually quite fast. The only criticism of it-- then there's another thing nice thing about it is that you can-- I'll show you what I mean by this. But you can sort of initialize this in ways that hop out of other local minima.

So let's say I just choose my initial conditions in the previous simulations. I just always just chose u0 to [? uN ?] to be some small random number. So the pendulum in the first thing would just shake here a little bit. And then it quickly changed until it swung up to the top.

I can pick x perhaps more intelligently. It won't have satisfied the constraints in the initial case. But I can choose an x. For the pendulum, let's say my initial guess at x would be a direct trajectory that goes straight up to the top.

If I start searching now for alphas that minimize this cost and satisfy those constraints, it just puts me in a different sort of area of the search space. And it might actually help me find the swing of policies. It's a very sort of heuristic thing to say. But it makes a big difference in practice I find.

So I think most people today actually use these direct collocation methods for trajectory optimization. Yeah, Rick?

AUDIENCE: [INAUDIBLE] change those parameters the vector would change.

RUSS TEDRAKE: So you're worried that this will change-- the constraint matrix would change?

AUDIENCE: Well, aside of the alpha [INAUDIBLE]

RUSS TEDRAKE: So I don't do it that way in my code. I do it, I say that this is valid from 0 to T over N, let's say. And this one is used from T over N to 2T over N and so on. So it just stretches out. So the dt is not constant. I use that same action for longer if my T stretches out. And that keeps the parameter vector constant.

But I think if you were to purchase a DIRCOL-- this is often shorthand as DIRCOL, direct collocation, DIRCOL. And there was a DIRCOL package that you could get in FORTRAN 10 years ago or something that I think a lot of people used. And I think those do things like it stretches out time. And then if dt gets ridiculous, it adds some more points. And then a more polished software package would do these things like adding parameters and then reinitialize-- reseeding the optimization. Mine doesn't.

Should I show you how it works? Is that the best thing to do here? It's a pretty simple idea. The part that I can't really express to you efficiently here is why that this is something that the solvers can do very well. I can tell you that it's about how SQPs do interior point methods, but I don't want to get into that.

So I think if you just sort of take it on faith that they're good at handling constraints, I think it's reasonable to think that maybe sending in an over-defined trajectory and allowing it to sort out the constraints is a reasonable thing to try. And in practice, let me tell you that it's pretty fast.

So can I do the pendulum DIRCOL now? So that one was fast before. Now you notice the time horizon here-- 3.06? Yeah, so that's what it wanted to be doing. It liked 3.06 better than two. So maybe my other ones would work better if I put 3 in there.

Can we do the-- someone called for the acrobot before, DIRCOL on the acrobot. Oh, I should turn off plotting. Sorry. Let's see what happens.

It's also got this final value constraint. That's why it quickly got to the goal to satisfy that constraint. And now it's making sure that all the dynamics that these trajectories are satisfied. And then, it's just optimizing within that constraint manifold.

I didn't really mean to-- I'm afraid to hit Control-C. It might crash. That's the one thing about [? SNAP ?] being a [? MEX ?] package calling FORTRAN-- stop. now my GUI is gone and my code is gone. Great.

OK, good. And it got to the top. No, that's the pendulum, sorry. That's cheating.

So turning off printing, just run DIRCOL for a second here.

AUDIENCE: How would using these methods can be extended to stochastic case?

RUSS TEDRAKE: I think they're heavily tuned to the deterministic case. I won't make a blanket statement saying they can't be extended. But even the feedback-- I really think the direct collocation in particular feels very specialized for opening the trajectory optimization.

This is slower than I remember. It gets there. Yeah, John.

AUDIENCE: Is there a reason why you need to use [INAUDIBLE] vector and not feedback [INAUDIBLE]. You could evaluate the u given the x and the [INAUDIBLE].

RUSS TEDRAKE: I think you could do that. I think the key thing is parameterizing the policy as well as x. But I think you're right. If you did some handful of w's or alphas there-- sorry-- then I think you could probably do that too, yeah.

I mean, implementing this constraint is the only real criticism that people have of collocation methods. Almost everybody uses them, it seems. The only criticism they have is that they have sort of this fixed-step integration in here. The constraint being satisfied-- it's hard to do sort of an ODE solver in that step, because you need to be able to take the gradients of your constraint.

So they tend to be fixed-step integration routines, roughly. And so, the accuracy-- if people point to a problem with collocation methods, they uniformly say that they're not as accurate as the shooting methods, because you don't actually numerically simulate your system carefully. You've picked some time discretization of the system, and you get it right for that. But I don't think that's a big deal actually. And if they're fast and they get out of local minima, then worst case, solve it this way and then do a little shooting method at the end to finish the optimization, if you like.

Run it one more time. John and I were talking before class that there's really no reason why you couldn't compute the gradients of an ODE, update inside here. Well, how would you do that? You do it exactly like we did the shooting method, right? You could sort of run a little-- to compute the gradients of your constraint, you could actually do the adjoint method or the RTRL method to compute those gradients. And then, that puts you somewhere in the land between direct collocation and multiple shooting.

I swear that my laptop must be operating on half a brain right now, because this was much faster in lab.

AUDIENCE: Do you have a power cord?

RUSS TEDRAKE: I'm not connected into power.

AUDIENCE: If you go to the battery, I guess it switches to energy saver.

RUSS TEDRAKE: Yeah, you're right. I could probably turn it off right now.

So that was a slightly different trajectory. That's just graphics though. I'd be exceptionally embarrassed if I plugged it in or changed the power settings and it was still slow. So let me just say that it was faster in live and we'll leave it like that.

So let me just make the point of, do people understand how a problem like this-- we sort of saw a demonstration that, if you could remember in your head what the acrobot did the first time and the acrobot did the second time. They both got to the goal pretty well. They took slightly different trajectories, at least to my eye.

So one of the complaints about any of these trajectory optimization methods is, they're only local methods. We're only going to find a local optima in my optimization. For me, I think about these problems a lot. It's still not completely intuitive what you think of a local minima in these settings might be.

There are some places where it could be pretty intuitive. So the pendulum, you can imagine if I found one policy that pumped up with one pump, you could imagine it might be hard to sort of get over a cost landscape, so you did two pumps to get up. That could be a case where a local minima makes sense.

I tried to come up with a little bit more obvious of an example here relating to the original grid world stuff we did. So now, this isn't so much a dynamics problem. But I thought maybe a path-planning problem would make the point.

So here's a random geometric landscape with Gaussian bumps that you try to avoid or you incur cost. I'm actually plotting the cost landscape as a function of x. So you see there's a small hill, which is trying to take me down to the goal in red.

Can I turn the lights all the way down for a minute here? It'll be dramatic this way. So there's a goal here in red. And let's say the initial conditions are over there in green. And your task is to take the system, which is x dot equals u, where x is the xy position of this, and u is the velocity in x, the velocity in y.

So just a trivial dynamical system, but you're trying to find a path that gets you to the goal with minimal cost. So I did this example just because some people care about this kind of example. But also because I think it's sort of critically obvious how you can have local minima.

If I get a trajectory that's on one side of the mountain and maybe the globally optimal trajectory is on the other side of the mountain, it might be hard for me to get across the mountain to that trajectory. So let's just see that happen.

So do direct collocation here, so what did I implement? OK, so I forgot to type-- so I just did direct collocation. And really quickly, it found this. So this is not the optimization software's fault.

What if I do this? So it found some nice path through the foothills here to the goal. Yeah, please.

AUDIENCE: Can you try to specify some of those-- since you have x parameterized, can you try to make it go through a particular set of bumps by specifying that?

RUSS TEDRAKE: Exactly. So the reason I chose to do direct collocation for this is, my initial guess, u was just some random vector. But my x was actually a direct line from start to the goal.

AUDIENCE: So if you had drawn it as a line between those first two and then going around to [INAUDIBLE]

RUSS TEDRAKE: So the one I thought to do was, let's just do-- just because it was easy to type-- let's do an initial x which just goes directly this way and then see what happens. I think that's what I had here. So if I change my x tape here is now linspace. So it interpolates in x0 straight to the goal. But then the other one is just x1 straight across, reading that code is not what you want to do in class here.

But if I set the initial x tape to be that and I run it again, It's doing its solving. It's properly doing it in some window. Oh, I turned off animation, didn't I? Oops, that was a failure.

But it found a different path. But it actually told me, Warning, exited. So I have this check page 19 of [INAUDIBLE] 6 the paper to figure out what the heck exit 41 means. But it basically couldn't satisfy the constraints. So I bet if I just run it again with the random initial conditions, it'll be OK.

But the point is exactly this. I still found one that just probably didn't satisfy some of the constraints at some small part of that trajectory here. It went the other way around the mountain. So there are local minima in these problems. In problems like this, it's completely obvious why there are local minima.

In the acrobot and things like that, you will find that there are local minima. If you start with different random parameterizations, you'll find slightly different swing-up trajectories. So our local minima-- a killer-- a lot of people say, well, that means these methods stink. They're subject to local minima.

I don't care if I'm in a local minima for the most part. I mean, if I was really going to have to walk around a mountain instead of walking through the mountains, then maybe I'd care. But if I'm doing an acrobot and it swings up like this instead of swings up like this, for the most part, I don't care.

So although people talk about it a lot, I find in most of the problems I care about, local minima aren't that big of a deal. They exist. Sometimes they can upset your numerics. But as long as you get to the goal, I'm happy.

Now, there are a couple other ideas in these trajectory optimizations. And on Tuesday, I guess, I'm going to wait till Tuesday now, first of all, I'm going to tell you how to stabilize these trajectories. Because that's useless as it is right now. If I just even simulated it with a different dt, it would probably fall or not get to the mountain.

But it turns out, even with a pretty coarse time step, if you stabilize the thing with feedback, then it works great. So I'll show you how to do the trajectory stabilization with an LQR method on Tuesday. And that's actually going to lead to another class of the trajectory optimizers, which would be an iterative LQR method.

And then, depending on how much I sleep this weekend, I was thinking about doing the discrete mechanics version of the trajectory optimizers on Tuesday too. We'll see.

So we'll push the walking back until Thursday just to complete the story about these trajectories solvers. Any Questions? They're pretty good. They're pretty fast, especially if you are plugged into the wall. OK, see you see you next week.