MITOCW | Lecture 8 | MIT 6.832 Underactuated Robotics, Spring 2009

RUSS TEDRAKE: So welcome back. I thought we'd start today sort of with a little bit of reflection, since we have covered a lot of material. Even though we've kept it to simple systems, we've actually covered a lot of material, and we're about to blast off into some new material.

So I thought, let's make sure everybody knows what we've done, roughly how it fits together, and where we're going, OK? So I've been trying to carry through the course two main threads. One of them is sort of the systems thread.

We start with pendula. We're getting to acrobots, cart-poles. We're going to get more and more interesting systems. In parallel, I'm trying to design this optimal control thread which tells you the way I think you should be solving these.

Along the way, I'm throwing in lots of puzzle pieces about partial feedback, linearization, energy shaping, things like this. That's because this is a hard-thought decision. I mean, this is a research class, really.

So I'm doing my very best to teach all this material to you as if there's a textbook on this. But in fact, there's no textbook. So what I've decided to do roughly is that, I'm trying to give you a very clean line of thinking through the optimal control.

But I do want to keep throwing in what other people do-- the domain-specific knowledge about acrobots and cart-poles and walking as we get to it. Because I think these puzzle pieces-- ultimately, there are things we can't do with optimal control yet. My guess is that ideas from partial feedback linearization are going to help.

I think ideas from energy shaping-- these kind of ideas are going to work together. So even though those aren't going to connect up perfectly in this class, what I'm hoping to do is give you all the pieces of a puzzle that nobody's actually solved yet, give you all the information I can give you about this class of problems so you can go off and write, well, final projects that are cited a hundred thousand times.

OK. So let me just make sure that's happening. So that's a tall order to sort of carry those threads. So let's make sure that's happening. Maybe I'll even use a whole board for it. So I think I've made no secret of the fact that I think optimal control is a sort of defining way to think about control for even these very complicated systems. All right.

So we've got one thread that we'll continue-- we'll get deeper and deeper in about optimal control methods. In particular, we've already talked about sort of two fundamentally different approaches to optimal control. We've talked about optimal control based on the Hamilton-Jacobi-Bellman equations, where we talked about the value function being described by a nonlinear partial differential equation. And then we also talked about Pontryagin's minimum principle, which you're probably all working on right now-- Pontryagin's minimum principle. These were two sort of analytical optimal control approaches, right?

As such, unfortunately, we only showed you much of anything on linear systems where we can actually solve those problems analytically. What were the big differences? Maybe I-- just to motivate this so you make-- so make sure everybody pays attention. I should also say a few things about what I hope you get out of the class, and for instance, what will be on the midterm when it comes around.

So I know that we're throwing lots of ideas out. If there's one thing that happens, one thing that you should be able to do off the top of your head is think about and talk about how these different tools that we're putting out relate to different problems. And for instance, if I were to give you a problem, you could make some reasonable guess at what kind of-- what methods that we've talked about might be most suitable for that problem.

The details of how you do a partial feedback linearization, I wouldn't expect you to absorb every piece of that. I would guide you through that on a problem, but with the expectation that you've worked through it once on a problem set and sort of have some ability to do that. But if there's one thing I want you to come away from this class with, I want you to understand the suite of tools we're talking about and have a sense for what you'd apply to what problem.

So in the Hamilton-Jacobi-Bellman equation, we applied it to-- we applied that-- which, remember, was this partial differential equation, with a hard nonlinearity, which describes the optimal solution. I should put stars on here to be here careful here. The optimal cost-to-go is described by that equation.

So one approach to these things is to directly try to compute solutions to this partial differential equation. We did it analytically for the quadratic regulator problems, the linear quadratic regulators.

Right. I didn't actually do it for the minimum time problem, because the minimum time problem, we know that the optimal solution in J is actually not-- its gradients are not well-defined across the entire-- for all x. That's the only reason I didn't do it for that. But for sort of smooth problems like the linear quadratic regulators, we could use these methods.

Pontryagin is a little bit more general. This is the one with the adjoint equation. So you defined the Hamiltonian is here some cost function plus Lagrange variable, Lagrange multiplier times your dynamics.

Pontryagin was more powerful. In some sense, we solved harder problems. We solved, for instance, the minimum time problem, for the double integrator, at least. The problem with it is just that it was too local. The Pontryagin ideas are based on a gradient statement of local optimality.

So we said along some trajectory, I can verify that if I change my control action a little bit along this trajectory, my cost is only going to get worse. So that's a necessary condition for optimality. That's when we can do a lot of things with-- but it's only going to give me a local optimality statement.

Are people sort of OK with those two methods we've been doing? OK. From the Hamilton-Jacobi way of thinking, we ended up with our first algorithm. Right. And we said that you could discretize the dynamics, and in the discrete dynamics solve whatever nonlinear problem you wanted, pretty much.

Right. And the reason we could do that is because these Bellman equations have this nice form that-- there's a nice recursive form. It was that J at some x, n is just the min over u. This is now-- maybe I should be very careful when I write it in discrete, because that's what we talked about it in min over actions A, discrete actions A, S A plus the J of S prime, where S prime is what I get for doing this.

OK. So if you sort of look at where there's white space left on this board, you might be able to guess what we're going to do next, our next big piece of the puzzle. We're going to derive our first set of algorithms now from the sort of Pontryagin methods. And we're going to talk about some policy search methods, the most important class being of trajectory optimization.

OK. Along the way, we've been sort of following this systems, we're developing these systems, right? In the optimal control, the analytical optimal control, we mostly just thought about double integrators. But those things could have applied to any LTI system. So that's supposed to be in line with that. I'll do my best to not move the board too many times in this.

We moved on to sort of pendulums, pendula. And we did the essential things. This is where I used to sort of tell you about dynamics. We talked about nonlinear dynamics, basins of attraction, all these things.

And then we did acrobots and cart-poles, where we talked about a lot of interesting ideas. We talked about controllability, we talked about partial feedback linearization, and we talked about energy shaping, even tasks-based. Lots of ideas in there.

Those ideas are my attempt to extract the most general. But the main topics of the sort of acrobot/cart-pole world, in acrobots and cart-poles, people talk about PFL heavily. They talk energy shaping, they talk about these kind of things. I only presented the ones that I think are going to be useful in general. But to some extent, right now you could think about these techniques as being orthogonal to our main line of thinking, OK?

Now, like I said, this is a puzzle that nobody has the answer to yet. So actually, what I want you to be thinking here is, what can we do with all these things? So for instance, I told you that dynamic programming works well for low-dimensional systems. Nonlinear systems, no problem. Low-dimensional, I can discretize the space, I can just run my algorithm, bup-bup-bup-bup-bup, compute the optimal cost-to-go optimal policy.

OK. But so maybe there's obvious things to do. And actually, I think there are obvious things to think about-- research questions that you could be thinking about for your final projects, right? So for instance, you know, we used partial feedback linearization to take-- at least linearize part of the dynamics of the system.

So this is wild speculation here, but let's say I have a problem that I could then describe as x1 dot is A1 x1 plus A2 x2 plus Bu then x2 dot equals f of x1, x2, u, where, let's say, the dimension of x2 is much, much less than the dimensions of the whole original system, which is what I get-- that's the result. That's the result I would get from doing a partial feedback linearization, where I have, let's say, for Little Dog, and I have five degrees of freedom and four actuators, that I'd end up with sort of one very nonlinear thing and then a bunch of very linear dynamics after doing a partial feedback linearization.

So fantastic research question-- so could I use that trick and combine it with dynamic programming, let's say, to use dynamic programming to solve the hard part and do some sort of LQR to solve the easy part? I don't know. Probably you can. I bet you can. And I'd be excited to think about, with any of you, you know, what you could do.

But these are the reasons I'm saying things like PFL, right? So imagine doing PFL plus dynamic programming. I'd bet you could do higher-dimensional optimization if you could exploit tricks like that. So that'd be a fantastic sort of research question to think about for your project.

Are people sort of OK with how this is going? Yeah?

AUDIENCE: Are any of those [INAUDIBLE] dimension that [INAUDIBLE] convergence or--

RUSS TEDRAKE: Good, OK. So they're not explicitly-- these are not explicitly targeted to solving optimal control problem. So proof of convergence, we have to define what we mean. I mean, PFL, certifiably, provides this sort of a dynamics.

The energy shaping, under some conditions, we showed we could regulate the energy of our systems. So there's-- each of these have their own sort of proofs and task space. But most of them are not directly aimed at proving that you've obtained some optimal policy.

AUDIENCE: If you obtain a kind of policy which would [INAUDIBLE] goal. If you have a specific state that you want to be in [INAUDIBLE] to get to that, maybe not optimally but--

RUSS TEDRAKE: Good. So in the energy shaping I talked about for the cart-pole-- the energy shaping plus PFL that I talked about for the cart-pole and swing-up, there's a citation in the notes of a guy that proved that for a set of parameters-- well, I'm being a little flippant-- there's a little-- there's a regime where that's guaranteed to work. But the general proof that you'd like to have is not there.

For the acrobot, I know about even less proof to that. There is one particular controller that we implemented that someone who took the class implemented before. And it does have a Lyapunov proof saying it'll get to the top.

But it sort of works by going-- this is an acrobots. It's going ee-ee-ee-ee. It's like really unattractive. So it does something really stupid. You wouldn't want to run it on your real robot, probably, unless you're very patient. But it has a proof to get to the top. So these are the trade-offs that people make.

OK. Right. So I hope that sort of-- I hope that was worth doing. I just wanted to quickly make sure we're all calibrated. So let's talk a minute now about-- so I said that for the dynamic program we showed it working on the pendulum. I put a big asterisk saying that there's discretization errors present.

So even for the pendulum, it'll solve it lightning fast. But it's solving the discretized system, not the continuous system. And the optimal policy you get out could be different than the optimal policy for the continuous system. OK.

So can you do dynamic programming for the acrobot and cart-pole? That's an obvious question. So the answer is yes. People have done it. My code on the acrobot and cart-pole runs in a few seconds. It's not a problem of dimensionality.

But the results are not-- of my code-- are not satisfying because of exactly the asterisks I put on the pendulum. So let's just sort of evaluate dynamic programming as we go forward.

So absolutely, the acrobot and the cart-pole both have four-dimensional state space, one-dimensional action space. Easily discretized these days. That's still low-dimensional enough that I can actually bin up the space. That wasn't true. When people weer doing it in the '80s, but it's true today. OK.

The only real problem with it is that there's discretization error. And essentially, the discretized dynamics can be a poor representation of the continuous dynamics. If you spend a lot of time with the acrobot, you find actually the acrobot's dynamics are pretty-- are sort of wicked in a lot of ways. So actually, spending a lot more time about it recently--

I mean, even just sort of making sure that energy is conserved when you put no torque into your system-- this is the basic absolute thing you do to make sure you got your equations of motion-- the acrobot, you have to put your integration tolerances up the wazoo to make sure you conserve energy. I mean, sort of [INAUDIBLE] resolution, the relative tolerance in the ODE solver in Matlab had to be something like negative-- 1 to the negative ninth or something to make this thing integrate and look like it had a flat line and energy as it's just swinging around with zero torque.

As a consequence, when you discretize it, and you run your optimal control which converges nicely on the discretized system, the same-- you can't discretize it with sort of 10 to the negative ninth precision. And you'll find that your discretized system doesn't conserve energy, for instance. So that's what's one of the major shortcomings.

OK. And so for that reason I'm going to move on in our-- when we're going to talk about the optimal control for the acrobot and the cart-pole, we're actually going to use some other methods. But I don't want to move on without seeding the idea that there are good ways-- there are ways to fix this, potentially.

So for instance-- I mean, I think there's lots of good work to be done in sort of the dynamic programming world. I think there are ideas from discrete mechanics and from finite element methods, where people have thought a lot about the consequences of discretizing PDEs and trying to, for instance, conserve quantities like energy.

My guess is if someone had some excitement or experience with these sort of methods, I'd bet the next time I teach the class I can say it works for the acrobot. So this would be a great final project, yeah? And publication.

Just do a smarter discretization of the dynamics, and then sort of-- so the discrete mechanics philosophy is that if you've taken your Lagrangian, and you've turned it into x dot equals f of x, u and then you discretize, then you've done-- you've already-- it's too late. You've already killed the beauty of the Lagrangian.

And the discrete mechanics point of view is you should discretize the Lagrangian, do discretization up here, and then carry that down to your equations of motion. And these sort of discrete mechanics principles tend to have much better properties and energy conservation and stuff like that.

We might get to it in our trajectory optimization family, but there's a line of work now called discrete mechanics and optimal control that's been done by Marsden and all at Caltech that I think that's my best lead right now and how to fix these problems.

OK. There's another idea out there for how to fix these problems. If you judge your problem is just that your discretization is bad, another big idea sort of is variable resolution methods, which says, let's stick to our guns, discretization is going to work as long as I have enough resolution. And because of computational limitations, I'm just going to make sure I put the resolution in the right places.

So if you discretize the pendulum or something like this, when you do your optimal control solution on this, and you find out, for instance, that when you're transitioning from this point-- this way, energy is not conserved. Or the value function at these corners are very, very different. So it looks like there's something more going on there, then let's just add more resolution there, until-- as much as necessary, sort of, to capture the dynamics.

There's a nice line of work in this by Munos and Moore, the same people I listed for doing the barycentric interpolation. They talked about variable resolution DP methods. I think Woody thinks that this is our-- this is the way to get value iteration to work on at least the minimum time problem for the pendulum and for the brick-- that if you use the right splitting criteria-- and that's the big question, I think, in this work is, what's the right splitting criteria-- then you can actually maybe make serious progress on these problems.

OK. So I've given you a line of thinking about one class of algorithms dynamic programming. We showed how they could apply to the pendulums and decided to not show how they don't quite work beautifully for the acrobot/cart-pole. If I build an algorithm that took overnight to run, then I think it works. But they don't run sort of in real-time in the class, so let's leave it as future work to make better to stay with programming algorithms for pendula and acrobots.

And I'm very, very serious about this. These are not killer problems. These are problems that-- I mean, these tools sort of that you see in the class, I think, just really haven't been put together before. I think that we're lining up all the tools to solve these basic problems. I wish they were all solved already.

We've just got too many problems and not enough time. But I mean, you could solve this problem in your final project and make serious contributions to the field. This is the state that the field is in. OK, good. So change the world. Do that for your final projects, ideally.

That's where we've come from. Let's start thinking about-- let me just bite off today the next big chunk, OK? You're going to finish this class with sort of a Chinese menu of tools that hopefully will help you solve all your problems.

OK. So analytically and computationally, there are two major approaches to solving these optimal control problems. Let's even just say computational optimal-- numerical optimal control. The first one, like I said, is you're trying to solve a PDE-- Partial Differential Equation. The particular name is the Hamilton-Jacobi-Bellman equation.

But there is a second approach. The second approach is policy search, direct policy search. Very much still is governed by the partial differential equations of optimality. But the solution technique is different.

Here's the idea. Let's design not directly the-- I mean, we've designed our cost function. We have our cost function, we have our dynamics. Let's not-- let's design a family of control systems with a bunch of parameters, OK?

So the policy search methods define some class of feedback policies that you care about that are parameterized by some vector alpha. We define a class of feedback policies. And then using the same exact formulations we used before, where we used J to represent the long-term cost of taking and starting with some initial condition at some time, which could be-- this is what I wrote down before.

Now I'm going to be even more specific and say, let's make J of alpha x0, t. I'm going to say it's a function of the parameters. And I'm going to say that u is now the cost of evaluating my control system with parameters alpha. So it's a little abstract right now, but let's make it concrete.

So here's a couple of potential parameterizations, right? So we talked about the linear family of feedback policies, linear feedback control with some big matrix K. Well, that's a perfectly acceptable policy parameterization. If I want to search over the class of feedback policies that are linear feedback policies, then I could call that pi of alpha x of t. It just happens that that control policy is alpha 1, alpha 2, alpha n times x.

It's a perfectly reasonable class of control policies. Actually, just to throw it out there, it's probably a bad choice, actually. Because I think people know that even sort of LQR problems are not convex in this parameterization.

I haven't told you how we're going to solve it yet, but let me just throw out the fact that I think most sort of serious control people wouldn't use this as a representation to search over, because it turns out the relationship to performance based on these parameters is complicated. Maybe unnecessarily so.

A much-- a very common parameterization is sort of an open loop control tape, I'll call it, where in the simplest form, let's say u is just is that a reasonable way to write it? And every time, I just-- this would be a zero-order hold. And at any time, I just find the closest sort of point in my control tape. And I'll put-- so I've got alpha at time 1, I've got alpha at time 2, alpha at time 3 just in the tape.

And I just-- as I run my policy, I ignore state, and just play out an open-loop tape. That's a perfectly valid policy representation. Maybe a better one would be something based on splines. Or even just a smoother interpolation, maybe that could be better.

People use things like neural networks as policy representations. There's a lot of work on things like radial basis functions. And in general, a lot of sort of kernel methods in machine learning you can use sort of-- the point of this line is you can use general machine learning function approximators.

And those tend to be reasonable policy representations, where maybe the weights in the neural network, even if you don't know what these things are-- I'm actually going to do a little bit of an introduction to function approximators once we start using them heavily in class. But just from seeing the words, even if you've never used one, you probably have a sense that these things are sort of ways to represent functions with a lot of parameters. And those are perfectly good candidates.

So the key idea here is, if we're willing to parameterize our control system with a class of some parameters, some finite parameters, then I can turn my optimal control problem into a simple parameter search. In general now, if I want to minimize-- the problem is to minimize over alpha, let's say J alpha from the x0 I care about at time 0.

I could describe J through those equations, through some Matlab function, let's say, and just say, find the minimum of this function. You can do it with sort of fmin or various-- any old tools from nonlinear optimization. Seem reasonable?

It's important to make sure we understand why it's different, OK? So-- do I still have this up on the board?

AUDIENCE: Are these [INAUDIBLE] approximators [INAUDIBLE] x as input and so y is output [INAUDIBLE]?

RUSS TEDRAKE: Potentially x and time as an input and u as an output. You're trying to-- the function approximators represent this function, right? They're mapping which depends on parameters alpha from x and t in the general case to u. in many ways, this is a very naive approach.

The dynamic programming view of the world is very beautiful. We turned our complicated long-term optimization of this function into a recursive form, where at each step I only had to think about my instantaneous control action. I did a min over u for that one step, and that was end to end, if I could solve my value function, then that was enough. I could use my value function to turn my long-term optimization into a short-term optimization, min over u.

Tell me if I need to say things differently. In many ways, this is the dumb approach. We're not-- we're throwing away the structure in the problem. We're just going to directly search over parameters. The saving grace is that I don't have to-- the value function can turn out to be a hard thing to represent, especially if-- with dynamic programming, I can't represented in 10 dimensions, let's say.

So this dumb approach can actually work in more complicated systems. The only problem is it doesn't guarantee global optimality. Like I said, in some ways it's a very naive approach. It tends to scale better-- it's not as sensitive explicitly to the dimensionality of the problem.

There's another nice thing about it, which there's no explicit need for discretization, which I told you was a big problem in the dynamic programming thing-- except for there's discretization potentially in the ODE sort of integrator. And that can make-- we do know how to make that arbitrarily accurate.

The only real killer of these methods is that they don't-- they're very subject to local minima. Yeah, please.

AUDIENCE: This function approximation, isn't it like discretization of your state space? [INAUDIBLE]

RUSS TEDRAKE: Not necessarily.

AUDIENCE: I mean, but you essentially-- you don't have full control [INAUDIBLE] available [INAUDIBLE].

RUSS TEDRAKE: It's a good question. So take the linear feedback example. If I have a problem that I know the-- if I take an LQR problem and I solve it with policy search, and I know the feedback policy should exist in the class of linear [INAUDIBLE] things, then I haven't lost anything by doing an approximation.

And in general, these things are-- so radial basic functions have the feeling of similar to discretization. But some of them are much smoother and much more continuous than these hard discretizations. And the way that you evaluate them, which is what's essential, is you're still going to find-- so if I evaluate the system by literally taking my parameters of my neural network, radial basis function, whatever, running this function without any discretization, then it'll give me an accurate measurement of this function.

Discretization comes into the-- doesn't come into the evaluation of the function. In dynamic programming, it's fundamental. Discretization is right there. We always operate directly under discretized system. So I do think these things are much closer to being continuous solvers.

You might say another disadvantage is that it doesn't exploit the recursion that we know to exist in the problem. So it sort of feels like we should be able to use that trick more generally. And a lot of times, these methods are going to be the very naive things which throw them away.

AUDIENCE: You said DP requires the discretized space?

RUSS TEDRAKE: Yep. That's what I said. Do you disagree?

AUDIENCE: [INAUDIBLE] can be [INAUDIBLE].

RUSS TEDRAKE: Well, then I would call that an approximate dynamic programming method, which is-- these are the-- it depends where you draw the line. I'm going to talk about those in the reinforcement learning part of the course. But the thing that I think people-- I think that we talked about, which has guaranteed results for the discrete system, which is sort of really dynamic programming, discretization is exactly fundamental.

So Alborz is pointing out that actually there are people that use function approximators in dynamic programming algorithms. And we're going to talk about those in the future. But they tend to be approximate. A lot of times, they have weaker guarantees of convergence. But we'll talk about those as they come up.

OK, good. So now we have a very simple problem. We've taken our optimal control problem that we've thrown all kinds of work into. And we've talked about the recursion, we've talked about the Bellman equation. And now we just said, OK, might as well just think of it, that-- if I run my robot with three different parameters, I'm going to get three different scores.

If I literally take my acrobot and I make the parameters all 1, then I'm going to get some score for running that policy. If I change my parameters so the third parameter is 2, I'll get a different score. And I'll get a different score if I run a different set of parameters.

I'm just going to run my trial for, let's say, for 10 seconds with different sets of parameters. And this is going to give me some landscape, which is J of alpha. Now typically in these problems, I'm going to be thinking about optimizing it from a particular initial condition. We can talk about later how-- if you want to get around that. But this is just some function J of alpha, right?

So how do we optimize J of alpha? Well, there's lots of good ways, from nonlinear programming, from nonlinear optimization. What are some good ways to find the minimum of J of alpha? Guess a lot of alphas, that's one approach. Pick the smallest one. OK.

AUDIENCE: You can form J in a way which is, you can take [INAUDIBLE] dynamics smooth then we can solve [INAUDIBLE].

RUSS TEDRAKE: OK, good. So let's say I have an initial guess at J, and I can actually compute the derivative of J. Which I can always do, because I could do it numerically if I wanted to, right? I can just evaluate it a bunch of times to do it if I had to.

But if I can compute dJ d alpha, then that'll tell me that slope. And I could, for instance, do gradient descent on that slope. I could do alpha-- my second alpha that I'm going to try is going to be my first alpha. I try minus some movement in the direction of the gradient.

I could take this, estimate the gradient, and then take a motion that moves me down the gradient, make a new update, move down the gradient, make a new update. And eventually I'll get to the minimum where the gradient is equal to 0. How many people have used gradient descent before in something? OK, good.

So nobody actually does that, I don't think, anymore. Because we have sort of-- I mean, that's absolutely the right way to think about things, and gradient methods are critical. But you optimization theory has gotten pretty good.

So there's another way to do it. Let's say I had-- I couldn't just-- I not only compute the first derivative, but let's say I could compute the second derivative. My initial guess here, that could be the first derivative. And it could be the second derivative.

Then what could I do?

AUDIENCE: Fit a parabola to it?

RUSS TEDRAKE: Fit a parabola to it? I didn't quite hear what you said. Is that what you said, too?

AUDIENCE: Steepest descent.

RUSS TEDRAKE: Absolutely. Let's fit a quadratic bowl to it, right? And actually, the problem I did right now, probably the quadratic bowl is a pretty good match to the real optimization. And why not move directly to this point and then fix-- find a new quadratic bowl and move directly to that point. So this would be a second-order method.

OK. And so Newton-- a lot of people call it the Newton method. Turns out it works just as well in high-dimensional systems. If I have a bunch of alphas, I can do these second-order methods.

And doing this, in general, is what is called sequential quadratic programming. Yeah. And they tend to converge-- there's an additional cost, potentially, of computing that second derivative. But you can do it by remembering the past-- the same way you can remember your-- you can estimate your gradient by remembering a couple of samples and just doing a numerical gradient. You can remember a couple of samples and compute the-- estimate the second derivative.

So I'd say probably the most common method used right now-- how could I say that? But one of the very common methods is to try to compute these analytically, because-- I'll show you a good way to compute those. And then these, which could be put could be potentially more trouble to compute, we'll just use sort of a numerical secant method to collect our second-order terms, and then use sequential quadratic programming.

The thing that makes sequential quadratic programming better than sort of the naive gradient descent is that it's faster. But the real thing is that optimization theory is just this beautiful thing. Now I can take constraints into account very simply. So let's say I have-- this is sort of a crash course in optimization theory. But I think you can say in a few minutes most of the key ideas.

I mean, to be fair, most of the lectures we've had so far, you could take an entire course on each one of those lectures. So pick your favorite, take another course. But, you know, I think that's-- I think it's useful to have the courses that go over a lot of topics, and that's what this is.

So what happens if I now have, if I want to minimize over alpha J alpha subject to some constraint, let's say-- I'll just call it-- I'm running out of letters here-- A of x equals 0. We know how to formulate those with Lagrange multipliers.

But in general, finding equality, solving for equalities, that's just root finding. That's actually no more difficult than finding minimals. I can use the same Newton method to find-- to do root finding.

So if I have some constraint, let's say, alpha-- oh, that was a really bad choice. Let's call this something other than A. Let's just call it f of alpha, just so I keep my dimensions in the same direction here.

OK. If I want to find-- I'd better make it go through 0. If I want to find the zeros of that solution, I can use the same exact gradient updates, right? I can define a zero crossing if I have an initial guess at the system. I take the linearization, its' going to give me a new guess for the zero point. I take the derivative there, that'll get me close. That's the Newton method for root finding.

OK. So by knowing the gradients, you could sort of simultaneously do minimization and root finding to satisfy constraints. Long story short, if you have a problem that has the form minimize alpha subject-- some function J of alpha-- it's potentially nonlinear, but you could take its gradients, let's say-- subject to linear constraints, equality constraints, or even inequality constraints, you can just hand that these days to some nice solver-- some sequential quadratic programming solver. The one we use these days in lab is called SNOPT-- Sparse Nonlinear Optimization Package something, I don't know.

OK. So you could start solving optimal control problems by literally saying, OK, if I run this-- just telling it J, telling it the gradients of J if you can. That'll make it faster. Even if you didn't, you could just say, here's J, find me the minimum of J. You hand it to SNOPT, it'll go ahead and do a lot of work and come up with the best J, which is going to be some minima of this cost function.

There's no guarantee that it won't find this one. It's subject to local minima. But sequential quadratic programming methods tend to be better than gradient methods in avoiding local minima, because, for instance, if I'm here and I estimate the quadratic bowl, if I just take bigger steps, then I tend to jump over some small local minima that a gradient method might get caught in.

So just experimentally, people know a lot about how it works on quadratic programs if they're actually-- if the system is actually quadratic. If it's a nonlinear system that you're approximating as quadratic programs, then they sort of wave their hands, but it works really well in practice. Yeah, OK?

So we have a new way of solving optimal control problems. Just write the function down in a function that SNOPT can call. Give it a set of parameters alpha, it'll churn away and find alpha. All that's left for us to do in this class is figure out the best way to hand it to SNOPT. We want to make SNOPT's computation as effective as possible.

And there's a lot of different ways to do it. So the first way is literally parameterize your control system, called SNOPT. But let's at least be smart about computing the gradients. Let's avoid asking our nonlinear solver to compute the gradients for us numerically, because we can give you those analytically, exploiting the structure in the additive equations, the additive cost optimal control equations.

And as it turns out, it's a direct and clear descendent from the Pontryagin minimum principle, OK? So I'm going to show you lots of examples of these things working on Thursday. But I thought today, let's just make sure that the basic idea of what we're doing here comes through, this policy search, and show you how to compute those gradients.

In fact, let me just tell you the result first. I think that works sometimes. OK, so given J-- I'll just leave off that end condition for now, the terminal condition. The goal is to compute partial J x0 partial alpha. My claim is I can compute that very efficiently by integrating the system forward from 0 to t backward from t to 0, and then I'll get my gradient.

OK. It integrates the system forward, just like you would do it, run any old simulation. But while you do it, keep track of a few key variables.

Similarly, g of x.

It's.

Anybody recognize that equation? It's written a little bit different form, but.

AUDIENCE: Filter equation?

RUSS TEDRAKE: It's not a filter. Well, it could be interpreted as a filter of something probably, but. It's an equation we've seen before.

AUDIENCE: Adjoint.

RUSS TEDRAKE: It's the adjoint equation from the Pontryagin. OK.

OK. Then the gradients--

OK. So I'm done writing for a second, let's talk. Do you remember the story from the Pontryagin? The derivation sketch I did, we said that we had some functional, right? It was that if we change our control actions, we want to make sure that changing our control actions at all doesn't increase the-- doesn't change the constrained minimization of J subject to the constraints of the dynamics.

y, in that derivation turned out to be the Lagrange multipliers that enforced the constraint. OK. What they did was they put the system-- by making sure that this equation was satisfied and this equation was satisfied, we made sure that we were at a stationary point, at a minima of our functional, our constrained functional, of our Lagrange multiplier equation. OK.

It's exactly the same reason we're doing it here. We now have a functional which depends on J. This functional J, the Lagrange multiplier functional. And the derivations are in the notes. I won't do it again. By going backwards, by going-- integrating forward, we ensure that this constraint is satisfied. By integrating backwards, we solve for the Lagrange multiplier.

What we're left with is we can now, since the gradient with respect to Lagrange multipliers is 0, the gradient with respect to the state equations are 0, the only thing left is the gradient with respect to the parameters alpha. And it turns out to be this sort of very simple equation.

So it's this beautiful thing right that actually-- I hope this is-- it's a lot to write on the board real quick, but it's actually a pretty straightforward algorithm for computing the gradients, efficiently computing the gradients partial J partial alpha. All I have to do is simulate the system forward, simulate this gradient equation backwards, and I'm left with a direct function alpha, OK?

How many people have worked with neural networks before? Yeah? OK. Well, this is the back propagation. This is the back propagation algorithm for neural networks. Turns out to be exactly the same. This is the continuous time form of it. People have worked on it and back prop through time for recurrent neural networks.

But the exact way-- the reason the back propagation-- so there was this revolution in the mid '80s about-- that basically suddenly, everybody said neural networks will solve any problem. some People still say that today.

The thing-- the only thing, really, that happened, I think, from my point of view, is that somebody came up with an efficient algorithm for computing the gradients of the neural network weights as a function of the input/output data. It's exactly this idea that you can march the system forward and then integrate backwards. In that case through a big neural network, you had to integrate these equations backwards.

Being able to compute those gradients faster was enough that the world started saying neural networks are going to match the computational intelligence of the brain and solve AI and all these things. So it's a little dry, maybe. But this is potentially very enabling to be able to compute gradients efficiently. It could change what problems you can solve.

OK. Are people OK with the big picture of where things are? Yeah? Good. So on Thursday, I'm going to show you, now that we know how to compute the gradients efficiently, I'm going to show you this put to work, the intuition of sort of changing a policy, searching in policy space to solve problems like the acrobot/cart-pole, and some simpler examples.

And the dumb idea is, let's just make it a straight, nonlinear optimization problem over alpha. And I'll try to help you compare and contrast the way that works compared to the dynamic programming. See you then.