MITOCW | Lecture 4 | MIT 6.832 Underactuated Robotics, Spring 2009

RUSS TEDRAKE: OK. Welcome back. Since we ended abruptly, I want to start with a recap of last time. And then we've got a lot of new ground to cover. So remember last time, we considered the system q double dot equals u, which is of a general form, just a linear feedback system, which is state space form looks like this, where it happens that a and b are particularly simple.

And we looked at designing-- let's not say designing controller-- we looked at reshaping the phase space a couple of different ways. The first way, which is the sort of 6.302 way, maybe, would be designing sort of by pole placement, by designing feedback gains possibly by hand, possibly by a root locus analysis.

So we looked at manually designing some linear feedback law, u equals negative Kx. And we did things like plotting the phase portrait, which gave us for q, q dot a phase portrait that looked like this, where this has an eigenvalue of negative 3.75 approximately, and this one had an eigenvalue of negative 0.25. This was all for K equals 1, 4. OK.

And we ended up seeing that just from that quick analysis we could see phase portraits which looked like this. They come across the origin, and then they'd hook in towards the goal. And similarly here. This one's so much faster that it would go like this.

Then we looked at an optimal control way of solving the same thing. We looked at doing a minimum time optimal control approach, not specifically so that we could get there faster, even though "minimum time" is in the name, because here remember, we could get there arbitrarily fast by just cranking K as high as we wanted, but actually for trying to do something a little bit smarter, which is get there in minimum time when I have an extra constraint that u was bounded, in the case we looked at yesterday was bounded by negative 1, 1.

And in that case when u was bounded, now the minimum time problem becomes nontrivial. It's not just crank the gains to infinity. And we had to use some better thinking about it.

And the result was a phase portrait which actually, I don't know if you left realizing, it didn't look that different in some ways. Remember, we had these switching surfaces defined here. And above this, we'd execute one policy, one bang-bang solution. And then below it we'd execute another one.

And the resulting system trajectories-- remember, this one hooked down across the origin and went into the goal like that. This one really did exactly the same thing, right? They would start over here. They'd hook down here with-- this time they'd explicitly hit that switching surface and then ride that into the goal.

So it's a little bit of a sharper result, possibly, than the other one. And that final surface was a curve instead of this line. And for that, we got to have good performance with bounded torques. Now, we also did the first of two ways that we're going to use to sort of analytically investigate optimality.

AUDIENCE: Can I interrupt you?

RUSS TEDRAKE: Anytime, yeah.

AUDIENCE: Was there a good reason we just-- basically said we want to do linear feedback there? Could we have done like x1 times x2?

RUSS TEDRAKE: Good, yeah. Because-- well, there's a lot of good reasons. So it's because then the closed loop dynamics are linear, and we can analyze them in every way, including making these plots in ways that I couldn't have done if this was nonlinear. Another answer would be that this is what 90% of the world would have done, if that's satisfying at all.

I think that's the dominant way of sort of thinking about these things. x1 times x2 is comparably much harder to reason about, actually.

AUDIENCE: I totally get that. But is there like a system that the optimal control that lies in the space that you have to take into account these different approximations.

RUSS TEDRAKE: Good. So this is an example of a nonlinear controller. It happens that the actual control action is either 1 or negative 1. But the decision plane is very nonlinear. So that's absolutely a nonlinear controller.

It came out of linear-- out of optimal control on a linear system. But the result is a nonlinear controller. OK. Now, certain classes of nonlinear controllers are going to pop out and be easier to think about than the broad class. But we're going to see lots of instances as quickly as we can.

OK. So we did-- we actually got that curve by thinking just about-- just using our intuition to reason about bang-bang control. At the end, I started to show you that the same thing comes out of what I call solution technique 1 here. I wouldn't call it that outside of the room. That's just me being clear here, which was based on Pontryagin's minimum principle.

Which in this case, is nothing more than just-- let's write it down, exactly what we mean by this cost function. We have some-- let me be a little bit more loose. We have J, some cost function we want to optimize, which is a finite time integral of 1 dt. That sounds ridiculous, but we're just optimizing time.

But we want to optimize that subject to the constraints that x dot equals f of x u, which in this case is our linear system; and the constraint that u was negative 1 in that regime; and the constraint that at time t, x t had better be at the origin. Given those constraints, we can say let's minimize T. We're going to minimize that J, sorry. I already got the t in there, so. Minimize with respect to the trajectory in x, u, that cost function.

I use this overbar to denote the entire time history of a variable like x t1 to t final, or something like this-- time t0 to t final. OK. That's how we set up the problem. It's just optimizing some function but subject to a handful of constraints.

Pontryagin's minimum principle is nothing more than putting Lagrange multipliers to work to turn that constrained optimization into unconstrained optimization. And for this problem, we can build our augmented system I'll call J prime here, which just is the same thing but taking in the constraints.

So first of all, we've got a constraint on x T equaling 0. So I can put that in as a Lagrange multiplier, let's say lambda times something that better equal 0, which in this case was just x t And then plus 0 to t1 plus the constraint on the dynamics, which I'll call it a different Lagrange multiplier p, times f of x, u minus x dot, this whole thing dt. Yes?

AUDIENCE: How do you impose the constraint that u is [INAUDIBLE]?

RUSS TEDRAKE: Awesome. Good question. So it turns out what we're going to look at-- we want to verify that this thing is optimal. So you might want to put that constraint right in here. But it actually is more natural-- here, let me finish my statement here. The way we're going to verify optimality of this policy is by verifying that we're at a local minimum in J prime.

I want to say that if I change x, If I change u, if I change p in any admissible way, then J is going to change. Small changes in here is not going to change this. I'm at a local minima in J prime. That's the minimum principle idea, right? I just want my-- if I'm at a minimum of function, the gradient is 0.

In the Lagrange multiplier, the minimum of this augmented function, the gradient had to be 0. So if I change any of these, I want that to be-- that change to be 0. So it turns out that the more natural way to look at this bound in u is by not changing-- not allowing u to change outside of that regime.

This is actually fairly procedural. So you end up doing this calculus of variations on J prime. But I actually-- I made a call earlier today. I think it's going to-- if I do it right now in the beginning in class, I'm going to lose you to-- I mean, I'm going to bore you and lose you. But it's in the notes, and it's clean. So I'm going to leave that hanging and let you look at it in the notes without typos that I might put up on the board, OK?

Because I want to move on to the dynamic programming view of the world, sort of the other possible solution technique. OK. So today, we're going to do-- you can think of it as just solution technique 2 here. And it's based on dynamic programming.

Now, the computer scientists in the audience say, I know dynamic programming. It's how I find the shortest path between point A and point B without reusing memory, and things like that. And you're exactly right. That's exactly what it is.

It happens that the dynamic programming has a slightly bigger footprint in the world. There's a continuous form of dynamic programming. OK. So a graph search is a very discrete form of dynamic programming. So I'm going to start with sort of-- I'm actually going to work from the graph search sort of view of the world, but to make the continuous form that works for these continuous dynamical systems.

And we're going to use this to investigate a different cost function, which is just this-- still subject to the dynamics, which in this case was the linear dynamics. OK. So before we worry about solving it, let's take a minute to decide if it's a reasonable cost function.

It's different in a couple of ways. First of all, there's no hard limit on u. But I do penalize for u being away from 0. So it's sort of a softer penalty on u, not a hard limit. And then these terms are penalizing it from being-- the system from being away from the origin.

And instead of going for some finite time and minimizing time, I'm going to go for an infinite horizon. So the only way to drive this thing, the only way, actually, for J to be a finite cost over this infinite integral, is if q and q dot get to 0, and you do u of 0 at 0. Otherwise, this thing's going to blow up. It's going to be an infinite integral.

So the solution had better result in us getting to the origin, it turns out. But I'm not trying to explicitly minimize the time. I'm just penalizing it for being away, and I'm penalizing it for taking action. Now, what's the name of this type of control? Who knows is? I think-- yeah, LQR, right? So this is a Linear Quadratic Regulator.

OK. It's a staple of-- it's sort of the best, most used result from optimal control. Everybody opens up Matlab and calls lqr. But you're going to understand it. Good. But to do LQR, to understand how that derivation works, we've got to do-- we're going to go through dynamic programming.

AUDIENCE: Couldn't we use the same cost function there as well?

RUSS TEDRAKE: Awesome. OK. So why don't I put that cost function down and just do Pontryagin's minimum principal? There's only one sort of subtle reason, which is that that's an infinite horizon cost. So I was going to say this at the end, but let's have this discussion now.

So this is an infinite horizon. Pontryagin's is used to verify the optimality of some finite integral. So let's compare-- well, I know you know value-- the dynamic programming. So maybe let me say what dynamic programming is, and then I'll contrast them. Yeah.

But the people sort of-- I just want to understand what happened. We got two different cost functions, two different solution techniques for now. And we're going to address in a few minutes why I did different solution techniques for the different cost functions. But I hope they both seem like sort of reasonable cost functions if I want to get my system to the origin.

Different-- we're going to look at what the result is, the different results. And actually, something I want to leave you with is that you can, in fact, do lots of different combinations of these things. You could do quadratic costs and try to have some minimum time. There's lots and lots of ways to formulate these cost functions. These are two sort of examples, but you can do minimum time LQR, you can do all these things.

OK. But with the way we're going to drive the LQR controller is by thinking about dynamic programming. And to do that, let me start with the discrete world, where people-- where it makes sense. So let's imagine I have a discrete time system.

So x of n plus 1 is f of x n u n. And I have some cost function. Now remember, in the Pontryagin minimum principle, which shows that there's a sort of a general form that a lot of these cost functions take in the discrete form, it's h of x at capital N plus a sum instead of an integral of n equals 0 to N minus 1 g of x n u n. OK.

Now, again, I said this sort of additive form of cost functions is pretty common. And you're going to see right now one of the reasons why. The great thing about having these costs that accumulate additively over the trajectory is that I can make a recursive form of this equation.

So in particular, if I-- so I should call this, really, what I've been calling J, that's really the J of being at x 0 at time 0. And I can compute J of being at x 0 at time 0 and incurring the rest of the cost recursively by looking at what it would be like to be at some state x at time N-- and that in this case is just h of x of n-- and then thinking about what it would be like at-- to be at some J of x N minus 1-- and that's going to be g of x n minus 1 u of n minus 1 plus h of x n.

Let me be even more careful. And I'm going to say, let's evaluate the cost of running a particular policy, u n is just some pi of J of x n.

AUDIENCE: Sorry, why is the first x a 0, and then the rest of the x's [INAUDIBLE]?

RUSS TEDRAKE: OK. So why did I put x 0 here? That was intentional. I'm trying to make x 0 the variable that fits in here. Here x is the variable that fits in here. But you're right, I could be a little bit more-- I should be more careful. So now J, a function of this variable x at time N should really just be h of x. Yeah, good.

So then this is-- I could say it this way. The other way I could say it is J x minus 1 equals x. Maybe that's the best way to rectify it. OK.

And when I'm evaluating the cost of a particular policy, I'm going to use the notation J pi here, say this is the cost I should expect to receive given I'm in some state x. To make it even more satisfying, let's just be the same everywhere. This is x 0, and here I'll say x 0 equals my x.

If I'm in some state x at time 0 executing policy pi, I'm going to incur this cost. If I'm at some state x at time N incurring this-- taking this policy, I'm going to get this. Here I'm going to get this. And even when I'm executing policy pi, I can even furthermore say that x n is f of x n minus 1 pi of x n minus 1. It's probably impossible to read in that corner.

OK. So you can see where I'm going with it. It's pretty easy to see that J pi of x at some N is just the one-step cost g of x n u n plus the cost I expect to see given that x n plus 1 at time equals 1. OK.

So the reason we like these integral costs or the sum of costs in the discrete time case is because I can do these recursive computations. And the same thing true if I look at-- if I define what the optimal cost is. So let's now define J star to be the cost I incur if I follow the optimal policy, which is pi star.

Well, it turns out the same thing works. But now, there's an extra term here. OK. So it's easy to see that the cost of following a particular policy is recursive.

It's more surprising that the cost to go of the optimal policy is equally recursive with a simple form like this, min over u. And this actually follows from something called the principle of optimality.

Anybody see the principle of optimality before? OK. It says that if I want to be optimal over some trajectory, I'd better be optimal over-- from the end of that trajectory. So if I want to be optimal for the last-- it's from n minus 2 to the end, then I'd better be optimal from n minus 1 to the end. So it turns out if I act optimally in one step by doing this min over u, and then follow the policy of acting optimally for the rest of time, then that's optimal for the entire function, OK?

OK. OK, good. So we've got a recursive form of this cost-to-go function that we exploited with the additive thing, the additive form. And now, the optimal policy comes straight out.

The best thing to do, if you're in state x and a time n. is just the arg min over u of g x, u plus J star x, n plus 1 n plus 1 with that same x, n plus 1 defined by--

So in discrete time, optimal control is trivial. If you have an additive cost function, all you have to do is figure out what your cost is at the end, and then go back one step, do the thing that acts-- that in one step minimizes the cost and gets me to the lowest possible cost in the future. And if I just do that recursively backwards, I come up with the optimal policy that gets me from any x in n steps to the end.

Does that make sense? Ask questions. Do people buy that? Is that obvious, or does that need more explanation? OK. Ask questions if you have them.

All right. So we're going to use the discrete time form again when we get to the algorithms. But I'm trying to use it today to leapfrog into the continuous time conditions for optimality.

So what happens if we now do the same sort of discrete time thinking, but do it in the limit where the time between steps goes to 0? So let me try to do the limiting argument to get us back to continuous time.

OK. Now we've got our cost function, again, is h of x at capital T plus the integral from 0 to T of g x, u dt. The analogous statement from this recursion in the discrete time is that J x at t is going to be a limiting argument as dt goes to 0 of the min over u of g x, u dt plus J x of t plus dt t plus dt.

OK. This is now-- that's just a limiting argument as dt goes to 0 of the same recursive statement. I'm going to approximate J x of t plus dt as-- this is J star let me not forget my stars-- as J star at x t plus partial J star partial x x dot dt plus partial J star partial t dt. It's a Taylor expansion of that term.

OK. If I insert that back in, then I have J star x of t equals the limit as dt goes to 0 min over u g x, u dt plus partial J star partial x-- x dot is just f of x, u, remember-- dt plus partial J partial t dt. And I left off that J x there, because that actually doesn't depend on u. So I'm going to put that outside here, plus J x and t.

Those guys cancel. And now I've got a dt everywhere. So I can actually take that out, and my limiting argument goes away. And what I'm left with, 0 equals min over u g of x, u plus partial J partial x star plus partial J partial t.

This is a very famous equation, will be used a lot. It's called the Hamilton-Jacobi-Bellman equation.

AUDIENCE: Russ.

RUSS TEDRAKE: Yes? Did I miss--

AUDIENCE: x dot in the middle term there.

RUSS TEDRAKE: Here?

AUDIENCE: Last equation. That x dot [INAUDIBLE].

RUSS TEDRAKE: Oh, thanks. Good. This is f of x, u. Good. Thank you. Good, thank you. That is the Hamilton-Jacobi-Bellman equation, often known as the HJB. So Hamilton and Jacobi are really old guys. Bellman's a newer guy. He was in the '60s or something.

A lot of people say Hamilton-Bellman-Jacobi. That doesn't seem quite right to me. That's some guy in the '60s sticking his name in between Hamilton and Jacobi. So I try to-- I will probably say HBJ a couple of times in the class, but whenever I'm thinking about it I say HJB, OK?

OK. So we did a little bit of work in discrete time. But the absolute output of that thinking, the thing you need to remember, is this Hamilton-Jacobi-Bellman equation, OK? These turn out to be the conditions of optimality for continuous time.

Let's think about what it means. So do you have yet a picture of this sort of what J is. J is a cost-to-go. It's a function over the entire landscape. It tells me if I'm in some state, how much cost am I going to incur with my cost function as it runs off into time. In the finite horizon case, it's just an integral to the end of time. In the infinite horizon case, I've started this initial condition, and I run my cost function forever. So j is a cost landscape, a cost-to-go landscape.

This statement here says that, if I move a little bit in that landscape in x, scale by this x dot, then the thing I should incur is that is my instantaneous cost. OK. The way my cost landscape-- the difference of being in initial condition 1 versus being in initial condition 2, if they're neighboring, goes like the cost function.

And there's the cost function-- the cost-to-go function lives in x, and it lives in time. OK. It's one of the most important equations we'll have-- Hamilton-Bellman-Jacobi equation.

AUDIENCE: So we can take out the partial case that [INAUDIBLE]? Because that one Is independent of u, the last term. So if we take that out, basically, the difference between the value to [INAUDIBLE] with respect to time, in this time and going to the next time that sort of seems like a TD error squared--

RUSS TEDRAKE: Oh, yeah. Yeah. Good. There's absolutely-- this is exactly the source of the TD error and the Bell-- yeah. It's exactly the Bellman equation. So yeah. So you're right. Partial J partial t could have been outside the min over u. It doesn't actually have u.

But we're going to see all those connections as we get into the algorithms. But for-- this now is a tool for proving analytically and driving analytically some optimal controllers. We need one more-- we need to say something stronger about how useful that tool is.

So there's the sufficiency theorem is what gives this guy teeth, OK? So I told you that the Pontryagin's minimum principle was a necessary condition for optimality. It wasn't necessarily sufficient. If you show that the system satisfies the Pontryagin's minimum principle, then you're close, but you actually also have to say it uniquely solves that, it's the only solution to that, solves the Pontryagin's minimum principle. So there's extra work needed. The theorem we're putting up here is this saying-- is going to say that if this equation is satisfied, then that's sufficient to guarantee that the policy is optimal.

OK. So given a policy pi x of t, and a cost-to go function, J pi x of t, if pi is the argument of this, if pi is the policy which minimizes that for all x and all t, and that condition is met, then we can-- that's sufficient to give that J pi x of t equals J pi of x of t and pi x of t pi star x of t.

OK. The proof of that I'm not even going to try. It's sort of tedious. It's in Bertsekas, if you like-- Bertsekas' book. But we're going to use this a lot. So if I can find some combination of J, pi, and pi that match that condition, then I've found an optimal policy.

OK. Let's use this to solve the problem we want-- the linear quadratic regulator in its general form. So they've got a system x equals Ax plus Bu.

And let's say I have a cost function J of x 0 is h of x, t-- the same thing I've been writing all day here-- g of x, u dt, where x 0 equals x, where h in general takes the form x transpose Qfx, and g takes the form x transpose Qx plus u transpose Ru.

To make things-- to be careful, we're going to assume that-- we're going to enforce-- we're choosing the cost function. We're going to enforce that this is positive definite, making sure we don't get any negative cost here. And similarly-- actually, it only has to be semi-definite. Q transpose equals Q greater than or equal to 0 and R transpose equals R. That one does have to be positive. Definite

OK. Here's a pretty general linear dynamical system, quadratic regulator cost. To satisfy the HBJ, we simply have to have that this condition-- so 0 equals min over u x transpose Qx plus u transpose Ru plus partial J partial x star times Ax plus Bu plus partial J star partial t, that had better equal 0.

So I need to find that cost-to-go function which makes this thing 0. It turns out the solution to these things, we can just guess a form for J. Let's guess that J star x of t is also quadratic, again with a positive-- it's going to have to be positive.

It could be-- in that case, partial J partial x is 2x transpose S of t. Partial J partial t is x transpose s dot t x.

OK. Let's pop this guy in. I want to just crank through here. So does it make sense at all, that the J of x, t would be a quadratic form like that? Why is that a reasonable guess? Yeah.

AUDIENCE: Because the final time [INAUDIBLE] match the [INAUDIBLE].

RUSS TEDRAKE: OK. So in the final time, that's a reasonable guess, because it started like this. Yeah. And it turns out-- I mean, we're actually going to see it by verification. But for the linear system, when I pump the cost backwards in time, this quadratic cost, it's going to have to stay quadratic.

OK. So I've got 0 equals min over u x transpose Qx plus u transpose Ru plus 2x transpose S of t-- bless you-- times Ax plus Bu plus x transpose S t x. I need that whole thing to work out to be 0 for the minimizing u.

So let's figure out what the minimizing u is now. Is it OK if I just sort of shorthand? I'll say the gradient of that whole thing in square brackets with respect to u here is going to be, what, 2Ru-- or u transpose R, I guess?

We're going to try to be careful that this whole thing is a scalar. We're always talking about scalar costs. So I've got vectors and matrices going around, but the whole thing has to collapse to be a scalar. The gradient of a scalar with respect to a vector, I want it to always be a vector. The gradient of a vector with respect to a vector is going to be a matrix.

So try to be careful about making-- that gradient better be a vector plus what's left here? 2x transpose S that guy there, right? But I have to take the transpose of that. So it's 2B transpose S of t. The S t transpose is not x--

I screwed up, sorry. It's still x transpose. I'm trying to-- x transpose S t B. That thing has to equal 0. And that's where I get my transpose back.

So u star, the u that makes this gradient 0, is going to be-- those 2's cancel. It's going to be negative R inverse B transpose S transpose x. Which is important to realize that was actually-- it's equivalent to writing negative 1/2 R inverse B transpose partial J partial x transpose.

OK. So what does this mean? So I've got some quadratic approximation of my value function. It's 0 at the origin always and forever. If I'm at the origin, I'm going to stay at the origin, my cost-to-go is 0. The exact shape of the quadratic bowl changes over time.

The best thing to do is to go down to negative of the partial J partial x is trying to go down the cost-to-go function. I want to go down the cost-to-go function as fast as I can. But I'm going to wait-- I'm going to change, possibly, the exact direction. Rather than going straight down the cost-to-go function in x, I might orient myself a little bit depending on the weightings I put on-- the cost I put on the different u's. So I'm going to rotate that vector a little bit.

This is what I can do, and this is the weighting I've done. So the best thing to do is to go down your cost-to-go function, get to the point where my cost-to-go is going to be as small as possible, filtered by the direction I can actually go and twisted by the way I penalize actions. OK. And it's sort of amazing, I think, that the whole thing works out to be just some linear feedback law negative Kx-- yet another reason [INAUDIBLE] to use that form.

OK. Sorry, I should be a little careful. This is-- it depends on time. So it's K of t x. Why should it depend on time? This is a-- what's that?

AUDIENCE: We switch.

RUSS TEDRAKE: Because we switch what?

AUDIENCE: The actuation.

RUSS TEDRAKE: There's no hard switch in the actuation here. This is saying, I'm going to smoothly go down a value function. This one isn't the bang-bang controller. This turns out to be a smooth descent of some cost-to-go function. Yeah?

AUDIENCE: The S t equals partial [INAUDIBLE].

RUSS TEDRAKE: I mean, S of t is time [INAUDIBLE] itself.

AUDIENCE: Yeah, so it [INAUDIBLE].

RUSS TEDRAKE: So intuitively, why should I take a different linear control action if I'm at a time 1 versus time 2?

AUDIENCE: Because you're time dependent. So if you're very close to the final time, you want to [INAUDIBLE] lots of control, because you don't have that much time [INAUDIBLE].

RUSS TEDRAKE: Awesome, yeah. This is a quirk of having a finite horizon cost function. In the infinite horizon case, it turns out you're going to just get a u equals negative Kx, where K is a variant of time.

But in the time-- finite horizon problem, there's this quirk, which is the time ends at some point, and I have to deal with it. If the bank closes at 5:00, if I'm here and it's 4:50, and the bank closes at 5:00, I'm going to-- I'd better get over there faster than if it was 4:30 and the bank closes at 5:00.

In my mind, actually, there's a lot of problems that are-- bank closing is a weird one, but there are a lot of problems that are naturally formulated as finite horizon problems. Things-- maybe a pick-and-place. The minimum time problem was a finite horizon, pick-and-place.

There are a lot of problems which are naturally formulated as infinite horizon. I just want to walk as well as I possibly can for a very long time. I don't need to get to some place at a certain time. OK. But in many ways, the finite horizon time ones are the weird ones, because you always have to worry about the end of time approaching.

OK.

AUDIENCE: How do we get S t?

RUSS TEDRAKE: How do we get S t? OK. Well, it's the thing that makes this equation 0. So what is that thing? I figured out what the minimizing u is. I can insert that back in.

So I get now 0 equals Q plus x transpose-- I'm going to insert u in-- K-- or I'll do the whole thing, actually-- S of t B R inverse times R times R inverse. So I'm going to go ahead and cancel those out. B transpose S of t x. And the negative signs, because there's two u's there. The negative sign didn't get me.

And then plus 2x transpose S of t Ax plus-- so minus B R inverse B transpose S of t x plus x transpose S dot of x. It turns out that this term here should be the same as that term there, modulo of factor 2.

If you look, it's S, B, R inverse, B transpose, S. So this one actually, I can just turn that into a minus. OK. And it turns out that everything has this x transpose matrix x form.

So I can actually-- in order for this thing to be true for all x, it must be that the matrix inside had better be 0. So it turns out to be 0 equals Q minus S t B R inverse B transpose S t plus 2 S t A plus S dot t had better be equal to 0. OK.

Now, I made some assumptions to get here. Know what assumptions I made? The big one is that I guessed that form of the value function. And one of the things I guessed about it was that it was symmetric. So let's see if we're looking symmetric.

So Q, we already said, was symmetric. That's all good. That guy's nice and symmetric. That's all good. So this is the one we have to worry about. Is that guy symmetric?

It's actually not symmetric like that. But I can equivalently write it as S t A plus A transpose S t, since S is symmetric. And that guy is symmetric.

I said a very strange thing. I just said that the matrices are-- this one is not symmetric, I can write the same thing as-- it's this. So what I mean to say is that these are equivalent for all x. Because this has got to equal this.

OK. So, good. OK. So this equation, which I'm going to write one more time since it's an equation that has a name associated with someone famous-- deserves a box around it, I guess. So this is the Riccati equation.

I'm going to move the S over to this side. It's a Riccati equation. And I also have that final condition that you rightly pointed out, where S of capital T had better equal Qf.

So direct application of the Hamilton-Bellman-Jacobi equation, I was able to derive this Riccati equation, which gives me a solution for the value function. Because it gives me a final condition on an S and then the governing equation which integrates the equation backwards from capital T to 0.

And once I have S, remember, we said that the u was just negative R inverse B transpose S of t x. So I've got everything. Once I have S, I have everything.

OK. So this is one of the absolute fundamental results in optimal control. It turns out that if you want to know the infinite horizon solution to the-- if you look at the solution as time goes to infinity-- remember, I wrote my cost function initially was-- the problem we're trying to solve is an infinite integral.

It turns out that the infinite solution is the steady-state solution of this equation. So if you integrate this equation back enough, it's stable. It finds a steady state where S dot is 0. And that solution when S dot equals 0, The S which solves this, that whole thing minus Q, is the infinite horizon solution. OK.

If you open up Matlab, and you type lqr A, B, Q, R, then it's going to output two things. It outputs K, and it outputs S. Solving this thing is actually not trivial.

So how do you solve that for S? The hard one is it's got this S in both places. But this is the Lyapunov equation again. It's so famous, it comes up so pervasively, that people have really good tools for solving it, numerical tools for solving it. So Matlab's got some nice routine in there to solve, to find S.

And when I call lqr with the dynamics and the Q, R gives me exactly the infinite horizon S and infinite horizon non-time-variant K. If you need to do a finite horizon quadratic regulator, then you actually need to integrate these equations yourself.

OK. I hate going that long with just equations and not intuition. So let me connect it back to the brick now. That was the point of doing everything in the brick world here. OK.

So we've got Q double dot equals u. We've got now infinite horizon J x is infinite horizon g x, u dt, where I said g x, u was 1/2 Q squared plus 1/2 Q dot squared plus 1/2 u squared. So now that's exactly in the LQR form 0, 1, 0, 0. B is 0, 1. Q is the identity matrix. And R is 1.

It turns out I can actually solve that one algebraically for S. If you pump all the symbols in-- I won't do it because there's a lot of symbols-- but in a few lines of algebra, you can figure out what S has to be, just because so many terms drop out with those 0's that actually there's-- There's the three equations and three unknowns. And it turns out that S has to be square root of 2, 1, 1, square root of 2. OK.

The u, remember, was negative R inverse B transpose B transpose S x, which, if I punch those in, gives me 1 square root of 2 times x, which gives me closed loop dynamics of x dot equals Ax minus BKx is equal to 0, 1, negative 1, square root of 2 times x.

OK. Now I'm going to plot two things here. First thing I'm going to plot is J of x. J of x is square root of 2, 1, 1, square root of 2. A little thinking about that, you'll see that it comes out to be an ellipsoid that is-- [INAUDIBLE]-- sort of shaped like this. I draw contours of that function, of that x transpose S x. And the cost-to-go is 0 here. And it's a bowl that comes up in this sort of elliptic bowl.

All right. So what is the optimal policy going to look like, given that that's my bowl? We said the best thing to do is go down the steepest descent of the bowl. I want to go down-- wherever I am, I want to go down as fast as I can.

But I can't do it exactly. That was actually sort of a-- that's OK. I mean, I can't do it exactly, because all I'm allowed to do is change-- I have one component that I'm not allowed to change, right? I have that my Q is going to go forward independent of u directly.

So B transpose S x is going to be give me a projection of that gradient onto this-- the thing I can actually control, which way I can point my phase portrait in that given my control. And then R is going to scale it again. And the resulting closed loop dynamics, let's see if we can figure that out.

So if I take the eigenvectors and eigenvalues that, well, it turns out I'm not going to make the plot. My eigenvalues were square root of 2 plus or minus i 1 over square root of 2 with v being 1 over square root of 2.

So the best thing I can possibly do is to go down that-- if I didn't care about-- if I didn't worry about penalizing R, I didn't worry about my control actuation, would be to go straight down that bowl. But because I'm scaling things by-- I'm filtering things by wearing what I can actually control, and I'm penalizing things by R, the actual response is a complex response which goes down-- goes down this bowl and oscillates its way into the origin.

OK, good. It was a little painful. But that is a set of tools that we're going to lean on when we're making all our algorithms. You've now seen a pretty representative sampling of what people can do analytically with optimal control.

When you have a linear dynamical system, and there's a handful of cost functions which you can-- either by Pontryagin or dynamic programming, the Hamilton-Bellman-Jacobi sufficiency theorem, those are really the two big tools that are out there. In cases, especially for linear systems, you can analytically come up with optimal control policies and value functions.

Why did we distinguish the two? Why did I use one in one place and the other in the other place? Well, it turns out the Hamilton-Bellman-Jacobi sufficiency theorem has in it these partial J partial x, partial J partial t. So it's only valid, actually, if partial J partial x is smooth.

The policy we got from minimum time has this hard nonlinearity in the middle of it. It turns out that the value function that you have in the minimum time problem also has a hard nonlinearity in it. If I'm here versus here, it's smooth, but the gradients are not smooth. The gradient is discontinuous. So on this cusp, partial J partial x is undefined.

So that's the only reason why I didn't lean on the sufficiency theorem completely. How did Pontryagin get around that? The sufficiency theorem is talking about-- it's looking at over-- roughly over the entire state space. It's looking at variations in the cost-to-go function as I move in x and in time.

Pontryagin, if you remember, was along a particular trajectory. It was verifying that a particular trajectory was locally optimal. And it turns out in problems like this in these bang-bang problems, along a particular trajectory, my cost-to-go is smooth. The cost-to-go in the minimum time problem was just time, right?

So the time I get-- the time it takes for me to go to here to here is just smoothly decreasing as I get closer like time. Along any trajectory, with these additive costs, the value function is going to be smooth. But along a non-system trajectory, some line like this, partial-- if I just look at J, how J varies over x, it's not smooth.

So Pontryagin is a weaker statement. It's a statement about local stability along a trajectory. But it's valid in slightly larger domains, because it doesn't rely on value functions being smoothly differentiable.

Now, for the first-order-- sorry, for the double integrator, the brick on ice, we could have just chosen our K's by hand and pushed them higher or smaller. We could do locus. We could figure out a pretty reasonable set of K's, of feedback gains, to make it stabilize to the goal.

LQR gives us a different set of knobs that we could tune. Now we could more explicitly say what our concern is for getting to the goal by the Q matrix, versus what our concern is about using a lot of cost in the R matrix. So maybe that's not very compelling. Maybe we just did a lot of work to just have a slightly different set of knobs to turn when I'm designing my feedback controller.

But what you're going to see is that, for much more complicated systems that are still linear-- or linearizations about very complicated systems, LQR is going to give you an explicit way to design these linear feedback controllers in a way that's optimal.

So we're actually doing a variation of LQR now to make an airplane land on a perch, for instance. We can-- we're going to use it to stabilize the double-inverted pendulum, the Acrobot, around the top. So it's going to be a generally more useful tool. Down at the brick, double integrator level, you can think it's almost just a different set of ways to do your locus.

OK. You have now, through two sort of dry lectures relative to the rest of the class, learned two ways to do analytical optimal control. One is by means of Pontryagin's minimum principle, one is by means of dynamic programming, which is through the HJB sufficiency theorem. And you've seen some representatives of what people can do with those analytical optimal control. And it got us far enough to make a brick go to the origin.

Right. And it'll do a few more things, but. OK, so that's about as far as we get with analytics. We're going to use this in places to start algorithms up. But if we want to, for instance, solve the minimum time problem or the quadratic regulator problem, for the nonlinear dynamics of the pendulum, if I take my x dot equals Ax plus Bu away and give it the mgL sine theta, then most of these tools break down.

Next Tuesday happens to be a holiday, virtual Monday. So we won't do it on next Tuesday. But next Thursday, I'm going to show you algorithms that are based on these. This is the important foundation that are going to solve algorithmically the same optimal control problems that we're-- more optimal control problems that we can solve analytically.

And then the-- we'll go on from there to more and more complicated systems.