MITOCW | Lecture 5 | MIT 6.832 Underactuated Robotics, Spring 2009

PROFESSOR: OK, welcome back. Sorry for the technical blip there. OK, so I guess lecture two. I challenged you. We talked about the phase space of the simple pendulum, and I challenged you to come up with a simple algorithm. I guess I didn't say simple, but I challenged you to come up with an algorithm to try to, in some sort of minimal way, change the phase plot of this system so that the fixed points that used to be unstable become stable and vise versa.

So today we're going to do that. I don't know if anybody do that for fun? Yeah, OK.

[LAUGHTER]

OK, so today we're going to do that. So yeah, the question is, can we use optimal control now, numerical optimal control, to reshape these dynamics, OK. And I want to start by doing sort of an evil thing but something that's going to make thinking about it a lot easier. We're going to discretize everything, OK. So let's start by-- we're going to discretize state, actions, and time, OK.

So I'm actually going to take my vector of x, which lived on the real numbers, and start thinking about integer number of states. I'll say what I mean by that. OK. And I'm going to take my actions, my continuous action space, which I've been thinking of as u, and I'm going to turn that into a discrete state space, a discrete action space. And I'm going to take time and turn it into some integer, discrete time, OK.

So and I'm going to try to be-- throughout the lectures, throughout the notes, I tried to be very, very careful to use X and U and time for continuous things and S for states, A for actions, N for discrete things. So we might find ourselves in situations where we have continuous state and discrete actions or some other combination, but that should be a code.

OK, so if we want to-- if we're willing to discretize state and time, then maybe one way to think about that on this picture is by thinking of every one of these-- this was my quick cartoon of the phase plot of the simple pendulum. Let's think about identifying each one of these possible states in the phase portrait as a particular state, OK. These little nodes, possible states we can live in. And through actions, we can transition to different states, if you see what I'm doing without drawing 100,000 circles here.

So let's tile the state space with discrete states. You could also think of it as drawing a grid and calling each box in the grid a state. And what that allows us to do-- we're also discretizing actions, so we have a finite number of possible options coming out of each state. It allows us to turn the continuous time optimal control problem into a simple graph search problem, OK. Graph search, we know how to do well. We're really good at that in computer science.

OK, so let's see how far we can get first by just thinking about this very non-linear, very dynamic thing on a graph search, OK. So we're going to do numerical optimal control. This is-- in particular, when people talk about the dynamic programming algorithm, they're often talking about discretizing state and actions. And we're going to use the standard optimal control formulation. I'm going to start with a finite horizon and say that my cost of being in state x, time t is h of x at the final time.

All right, this is the continuous time optimal control. And I'm going to start thinking of that now as being in state S at integer time N and having me be at some final cost on S plus a sum from N equals 0 to N of g SA, OK. And my dynamics now are going to be of the form S-- maybe I should even write more explicitly, S N plus 1 is a function of SN, AN, OK.

OK, so again, dynamic programming exploits the fact that you can write this in a recursive form. So if I want to find the optimal cost to go, which I'll call J star, at the final time, it's just h of S, right. And going backwards in time, this is just going to be the min over a of g S, a plus h of S prime, where S prime is. Right? I'll get one-- if N is-- N minus 1, I get one of these, and then I get the final cost, OK.

And going backwards, we have this recursive form, which is min over a g S, a plus the cost to go from S prime and n plus 1 using that same S prime. OK, I want to make sure you see why that is, why this-- this is magical, right? The fact that I can summarize my optimal cost to go by doing a min over a single action, that's really magical.

Just to make that extremely clear, think about J star at N minus 2, let's say. So I have to minimize over two actions. I have to minimize over, let's say I'll call them a1 and a2. I have two steps left to go.

So I have to minimize S at a1 plus g of S prime, let's call it, a2 plus h of S double prime. That's my minimization that I'm trying to solve in order to find the optimal cost to go, where S prime is f of S, a. S double prime is f of S prime. This is a1, and this is a2. I'm just expanding this sum for the last two g's.

And the cool thing is that, because of this additive form of g, this term doesn't depend at all on my decision a2. I'm given a current state S, and I have to decide my action a1. Nothing about this term depends at all on a2, OK.

In contrast, this one does depend on a1, because S prime depends on a1. This one depends on a1 and a2. This one certainly depends on a2. You see what I'm saying?

So I can rewrite this as min over a1 f of S a1 plus min over a2 g of S prime a2 plus h of S double prime. I could just move that min inside to the only terms that matter. This is intended to be a moment of clarity, and I don't see a clarity on your faces. Does that make sense, that this doesn't depend on a2?

I know I'm going to- a1 is my action at time N minus 2. a2 is my action at N minus 1. The action I take next time has absolutely no effect on my current state or my current action. So the great thing is this here is just-- this whole term right here is just J star of S prime at, I'm calling it, N minus 1 here.

So it's really the fact that we're taking this min over this additive form that allows us to write the recursive statement like this that says, the best thing I can do with additive cost and all these things is to, in a single step, take the action which minimizes my one step cost combined with the cost I'm going to get from being in the state I transition to for the rest of time. It's a magical thing. At whatever time I'm at, I only have to think one action ahead if I've already got my J star computed, OK.

Simultaneously, it's saying that I can compute the optimal cost to go. I could compute the optimal-- I know exactly how much cost I'm going to incur from any state, given I follow the optimal policy, if I just work backwards in time. And when I'm in time N minus 1, I don't have to think about the actions I was going to take beforehand. As long as I know what state I'm in, because that state encompasses every action I've taken in the past, that state contains all the information, all I have to think about is the last action I'm going to take to decide my optimal policy one step from the end of time, OK.

So the fact that you can solve these things backwards in time, that's the principle of optimality, OK. Ask questions if you don't like what I said. I think that the graphics that are about to come are going to make things clear, too.

OK, so what does that mean? What are the implications of that? All right, for the additive costs, I can compute J star recursively from the end of time, which, in this case, is N back to 0.

And the optimal action, the optimal policy, which I then want to call pi star, which could in general depend on the time, is just argmin over a. It's the action which minimizes that same expression. So I can compute J star recursively backwards in time, and if I know J star, then I essentially know my optimal policy. I know the best action, OK.

So but for this reason, the fact that the cost to go, the cost I expect to incur given I'm in state S and I'm running from time N, the cost to go becomes a very central construct in optimal control. All right, so part of the goal for today is to give you some more intuition about J star, OK, because it's actually a very intuitive thing, but you can be lost, I think, in the equations.

So let's give you more intuition about that. I'm going to do that by getting a little bit more abstract, well, simultaneously abstract and concrete.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Because it's finite horizon.

AUDIENCE: You know that the reward function is dependent on time.

PROFESSOR: I haven't included that. You can make the reward function depend on time. But even if the reward function, or cost function in my world, is-- there's a difference between optimal control people and reinforcement learning people.

The optimal control people are pessimists. Everything's a cost. And the reward reinforcement learning people give rewards out. So I guess I'm a pessimist.

So yeah, so my cost is actually not a function of time. I could have made it that. But because there's a finite horizon time, that means my policy and my cost to go function still depends on time. Because if time ends in one step, I'm going to do something different than if time ends arbitrarily far in the future.

OK. So we're going to-- my goal here is to get intuition about cost to go and dynamic programming, which I'm often going to call DP, OK. And I'm going to do it with the grid world example. This is right out of the reinforcement learning books.

OK, so in that pendulum phase plot, I discretized the state space, and I started talking about transitions between states, OK. I can make that even more transparent by saying, OK, now you're a trashcan robot in a room. You're going to be in one of these tiles. You're on one of these blocks, so there's a finite, discrete state space, OK.

I won't draw a trashcan robot, but let's say I'm here. And when you're here, you have five discrete actions you can take. You can move up, you can move right, down, left, or you can sit still. OK. And discrete states and discrete time. Every time you take an action, in the next time step, you'll be in the next grid box.

OK. Let's say I've got a goal state somewhere in the world. Well, we can formulate plenty of good optimal control problems to get us to that goal state. So plenty of good cost to go functions in the additive form-- let's say I want to do minimum-- I want to get there in the minimum time.

Well, then I can just set g of S, a to be-- to actually have it in units of time, I should put a 1 if S is not at the goal and 0 if S is in the goal, OK. And I don't actually care about actions. I have five discrete actions I can pick from whenever I'm in a state. If I'm not at the goal, I'm going to incur a cost of 1.

So it's in my best interest as a trashcan robot to get to the goal. If I'm minimizing that cost, I'm going to get the goal as fast as possible. And actually, the units, the cost to go will tell me the number of steps to get there.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Right. So I'm going to do that graphically. But let's say there's a finite horizon now, but this is how I'm going to get to infinite horizon, so. And let's say that h of S is just 0. I don't really care where I am at the end of time. Or I could have h of S be this same function. That would be fine, too.

OK. How's is it going to look? What is J-- well, let's be specific about h. Let's make h actually be the same as g here.

So I'll say it's g S with the 0 action. So since this doesn't depend on actions, it doesn't matter. Let's say h is the same function as g there.

So what does my cost to go look like at time N? My optimal cost to go given I'm in some state, and it's time N. This is a function over S, and I'm time N. And what is that function?

AUDIENCE: g.

PROFESSOR: Yeah. Well, if I'm not in the goal, it's that. It's the same as g, or h in this case. OK. What does g star of S N minus 1 look like? Now I have time to take one action, OK. So--

AUDIENCE: One step away from the goal is 1. If you're on the goal, it's 0, but anywhere else, it would just be 1.

PROFESSOR: Awesome. Right? If I'm on the goal, I can do nothing, incur zero cost to go. So the best thing for me to do if I'm on the goal is to stay there, OK. If I'm a long way from the goal, then I'm not going to get to the goal in two steps, so I'm going to incur two units of cost. I'll say loosely far from goal.

And then there's this in-between place, which is if I'm one step away from the goal, I can take the right action and get there and incur only one unit of cost. All right, what's it going to be-- what's J S N minus 2 going to be? It's going to be 3, 2, or 1, depending on how closely-- if I'm near the goal, I've got a chance of getting to the goal and stopping this insane adding cost.

Stop the madness. Get to the goal. Otherwise, I'm going to just incur the cost no matter what I do, OK.

So what's the optimal policy? If I'm on the goal, what's the best-- the best action to take is to sit still. If I'm one step away from the goal, the best thing to do is to move to the goal, whether it's up, down, left, or right. What if I'm out here? What's the best thing for me to do?

Doesn't matter at all. I can do anything I want. I'm still going to incur the cost, so you might as well just choose your policy at random, OK. So optimal policies aren't necessarily unique. Sometimes multiple actions are equally optimal.

OK, here's your world. I have put the goal always at 2,3, just randomly, OK. You are a blue star. The goal is a red asterisk. It's a-- take you back to the '80s or something, video games. OK.

So let's just very simply-- I'm going to run this value iteration algorithm on it, OK, and I'm going to plot, at every step of the algorithm, the cost to go, OK, and the policy, actually. So it's not going to be-- I have my more general value iteration code that's not going to be quite as beautiful, but--

[TYPING]

OK. Well, that went pretty fast. There was supposed to be pause there. Let me get that-- add a pause in there quick, but-- OK.

Here is J at time-- at J at capital N. My cost function is 0 if I'm at the goal, 1 everywhere else, OK. My policy, it doesn't matter what I choose. I've actually chosen to do-- I didn't put this-- I didn't give you a key, but 0 is the do nothing action, OK.

So this just has do nothing everywhere. This is the lazy policy, I guess. And the cost it's going to get is it's going to get no cost if it's at the goal, one cost if it's everywhere else. OK, if I'm now computing J S N minus 1, you guys told me what that is. That says it's 0 here, it's 1 here, it's 2 everywhere else, right.

And the co-- now you can see my key here. Orange must mean move down, red must mean move to the left, green must mean move to the right, and so on, OK. The value-- this backwards propagation, this dynamic programming propagation is a very beautiful and intuitive thing, OK.

Every time I take a step, a few more states become reachable. In that amount of time, I can get to the goal. The resulting cost to go function is simple. It's just the distance, the number of cells from the goal, yeah.

And the policy, again, it's not unique. But this one, just because of the ordering I chose, and I just do a min over the actions, says it's always going to move down in that orange area, it's always going to move up in the blue area, and it's just going to-- so that's one of the optimal policies, all right.

Now Alborz asked a good question, what's my horizon time? So I'm actually just working backwards from some arbitrary capital N and just going backwards in time further and further. But it turns out for this problem, and for many problems, everything converges, OK. After some amount of time, the optimal cost to go stops changing, and I know that's my optimal policy. Walk down.

And this is too simple. This is painfully simple. But I think that intuition is going to take us a long way with the value methods, OK.

AUDIENCE: So, Professor?

PROFESSOR: Yeah.

AUDIENCE: In this example, the optimal policy is not unique.

PROFESSOR: The optimal policy is not unique. The guy could have just as well gone left first and then down. So how does that manifest itself in those equations?

There's multiple min over a's. There's multiple a's that give me the same J star S and N minus-- or plus 1, whatever. Multiple actions give me the same long-term cost, so I could equally pick any of them, yeah?

OK, to make a more careful analogy to the more continuous world, that was a perfectly good minimum time problem. I could have equally well chosen a different cost function. Oh wait, let's put the obstacles back in, all right.

So the cool thing is obstacles aren't going to make it any harder for us to solve this problem in our head. It's a nice observation that they don't actually make it any harder for the algorithm to solve it either. And that's a general principle. That's something I definitely want you to get out of this course, is that when we're doing analytical optimal control, every piece you add to the dynamics makes things cripplingly difficult. And so you have to stay with these very simple dynamical systems.

OK, the computational algorithms are actually pretty insensitive to how complex the dynamics are. They're going to break down in a different way, OK. So there's these different tools for different-- that are good for different problems. And there's a lot of problems which are very amenable to these computational tools that people aren't-- I mean, you can solve brand new problems pretty easily with some of these algorithms.

OK, so let's think of another cost function. Let's do the equivalent of a quadratic regulator. I just had that whole spiel and forgot to run the boundary-- the obstacles together in Soapbox.

OK, so now I'm just going to put in some obstacle. And if you see-- whoops, sorry. If my state-- OK, so I promised to use S and a in my notes and on the board, but I guess I didn't do it in my code.

Sorry. So x equals the goal, then the cost to go is-- the cost, instantaneous cost, is 0. Otherwise it's 1. If there's an obstacle, I just give it a high cost of 10.

So if I put that obstacle function in there, then I've got my same 0 cost for the goal. I've got a 1 cost almost everywhere, but I've got a 10 there. That's my cost function. And as I backup, a couple of things happened.

First, this thing quickly figures out how to get off that obstacle as fast as possible and decides not to go there anymore. And then as you back up the cost function, the colors are a little more muted because I have this high color here. But the same basic algorithm plays out until it covers the space. And my s-- oh, that was a--

[LAUGHTER]

--lucky initial condition. OK, good. Now he has to go around. Wow. OK, so adding an obstacle in the grid world is clearly trivial. It's nice to think that adding an obstacle when I get back to the pendulum would be trivial, because that's not trivial for most of your other control derivations.

OK, so minimum-- the quadratic regulator now. Now here, the cost I want is x of u, in the continuous world is some x minus x goal transpose Q x minus x goal. And you have to map that down into the integer world, the states. There's not a particularly clean way to write that, so I'm just going to allow you to imagine that it's trivial to code. Imagine that transition.

OK, now my cost function is just penalizing me for being away from the goal. But it's not a 0 and 1. It's penalizing me more smoothly for being away from the goal. So what's the best thing to do? The best thing to do is still to get to the goal as quickly as possible. It actually doesn't really change the optimal policy here, but it's a more smooth cost function, which, in some problems, gives you nice properties.

It turns out the optimal policy is more unique in this case. But that would have been an optimal for the minimum time problem, too. And it converges nicely and goes to the goal in the same way, and works fine with the obstacle, of course. OK?

Good. So now you have a little bit more intuition to work with on these cost to go functions. A couple of important things happened there that I want to highlight. First of all, I really want you to think in terms of cost to go functions.

They're really intuitive. The cost that I will obtain till the end of time, the optimal cost to go says if I'm acting optimally, this is the cost I'm going to incur. And the optimal cost to go gives me the optimal policy, OK.

And just to calibrate you here, J star is called the optimal cost to go, but it's also sometimes called a value function, optimal value function. A bunch of different communities talk about the same things with different words. These are the optimists. These are the pessimists.

OK, the other thing that we saw is that for many problems, the limit as N goes to negative infinity-- I know that's a silly thing to say, I guess, but-- that a lot of times this thing actually goes to some well posed J star. It doesn't have to. Sometimes it blows up.

Another way to think of this is that I said J S of N is S of capital N. It's the limit of this as capital N goes to infinity, if you think of it in the forward way. So in order for this thing to converge to some nice solution, this sum had better converge in the limit. For my choice of g for the minimum time problem, and for the quadratic regulator, both of these had the property that when you get to the goal, you stop incurring cost. So that integral-- as long as you can get to the goal, that integral-- the sum, sorry, is going to converge.

If I had chosen that I give a cost of 1 when I'm at the goal and 2 when I'm anywhere else, then it wouldn't have converged. The cost to go would have gone to that same shape, but then that shape would have just kept increasing every time I go farther back in time. That whole function would just move up by one every increment of time, OK.

But for a lot of problems, we do have this nice limiting behavior, OK, and that gives rise to the infinite horizon problems. So so far, I had talked about finite horizon, but a lot of time, a lot of problems we write as infinite horizon. OK.

When your problems are infinite horizon, J and J star don't depend on time anymore. And the optimal policy doesn't depend on time. So J star and pi, all these things are just functions of S, not of time, OK. And for these to be well posed, that sum had better converge.

Now just to say it, but not to dwell on it, a lot of people do write other formulations that handle that. For instance, a lot of people do discounting. A lot of people like to solve problems of this form, OK, just to make it more likely that that sum's going to converge, for instance. And there's some problems which really do have discounting. Yeah.

AUDIENCE: So that's less than 1 [INAUDIBLE].

PROFESSOR: Yes, thank you. Good. Good call. Thank you. OK, so you know the basic dynamic programming equations, no? Let me just say one word about implementation, if you want to go home and make your own '80s graphics game in Matlab.

For discrete states, discrete actions, J, even J star of S at some N, it's a vector. Typically I think of it as sort of a dimension of S by one vector. And dimension isn't the right word. This is-- so the cardinality of S, let's say, something like that, a big S, the number of possible states by one vector.

And it's very practical to write that recursion for all states as a vector equation. So if I think of J star as being a vector, I have to do a min over a of g S, a. But g is another vector which depends on a. It's an S by 1 plus-- I can write it as a vector equation where this is a vector. This is a matrix. This is the transition matrix.

And this is my vector again. And then transition matrix is just 1 if f iA equals J and 0 otherwise. OK. That's just a standard graph notation. So it's trivial to code these things in Matlab with just a bunch of matrix manipulations.

OK, we understand everything about the grid world. I think it is a very helpful example, actually. Now let's think about the more continuous problems that I care about.

What if, instead of having the dynamics of this moving left, right, whatever, my dynamics, my transitions came from my equations of motion from one of the systems we care about? So let's think about the double integrator. q double dot equals u. Let's do the min time problem. I can use the same minimum time cost function I did before, OK.

[TYPING]

OK. This one, I didn't leave the pause in there, but look what happens. Oops, sorry. Meant to do that. Make it bigger.

I pop the same-- let me turn the lights down. I pop that same exact set of equations. I run the same value iteration algorithm, dynamic programming algorithm. I should have said, people tend to call it value iteration for when you take the infinite horizon version and dynamic programming if you call it-- if you do the finite horizon, but they're exactly the same thing, OK. So I might accidentally say value iteration because I'm used to it.

OK, so I took my double integrator dynamics. I discretized my space. I made my cost function exactly the same as the minimum time cost function I used in the grid world, where there's a 0 cost of being at the goal and 1 everywhere else. And look what pops out.

This is the cost to go function, is a function of state, and that's the policy. Remind you of anything? Right?

Now I've got a big disclaimer that goes at the end of the lecture, but for now, let's just say that that's the perfect solution. The discretization is going to make this thing a little bit wrong. I'm going to say a few things about that at the end of the class.

But the cool thing is that I pop my cost function in. I pop my continuous dynamical system. It's discretized. [CLICK] Run dynamic programming. As I back it up, it converges to some-- as N goes back, it does converge for this. It was the minimum time problem.

And I get my optimal policy out, which is a bang bang policy, which is decelerate when you're at the bottom, accelerate when you're at that top. And that switching surface shows up in green just because it's interpolated. But when you know it, that's what we know about bang bang controllers, OK. Yeah.

AUDIENCE: Did you have to encode that your only three actions were full forward, full backward, and--

PROFESSOR: The minimum over a is always going to choose the rails. In fact, in this implementation, they could have chosen in between things, and that's what it did right on the switching surface because of some-- it chose 0.

AUDIENCE: OK, so you left the general just as--

PROFESSOR: I left it general. Yeah. So always, when I discretize the state and I discretize the actions of these continuous problems, I'm left with a finite set of states, a finite set of actions. So it can't pick unbounded. It's fundamentally bounded in actions that it can choose, and it chose those bounds.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Say it again.

AUDIENCE: How do you define the transition model?

PROFESSOR: Good. I'm going to say some words about that in a minute, too, OK. Yeah. But not yet. Just give me a minute.

OK, let's say we want to solve the LQR problem, the quadratic regulator cost for this.

[TYPING]

So I animated the brick for you just to keep it exciting. OK, so what pops out? This beautiful quadratic cost to go function, OK. Now this is a little bit off. It's supposed to be a linear function. It almost is, but there's some saturation because of my actuator limits, OK.

But within the resolution of sort of my discrete actions, that's what we expected, OK. So I can do this for the brick. I'm going to tell you the caveats again in a minute, and I'm going to tell you the interpolation in a minute. But first I just want to help you realize that this-- we can pop these equations in if we're willing to discretize the state and action space. Even for pretty hard problems, I can just [CLICK] let it go. It's pretty fast, too, actually.

OK. So now why not-- analytically, we had a hard time doing the pendulum, those nonlinear equations, OK. But if we tile the space, turn it into a graph, then I can run the exact same algorithm on the simple pendulum, OK. So let's do that.

[TYPING]

What am I going to get here? So minimum time for the simple pendulum. I've got my pause back in here. It's hard to see, but there's actually-- it's 1 everywhere except for 0 at the goal, which is the-- now I'm in phase space, so that's pi at 0. That's my unsteady fixed point, OK.

I've got a blue 0 there, 1 everywhere else. At the end of time, my action is just do nothing, because there's no benefit to doing anything. And as I back up in time, this will give you a key to what-- you can see a little bit about my interpolation as I do this.

OK, then it starts giving me incentive to move. Again, when you can't get to the goal, that's actually just noise there. But this thing quickly figures out-- oops. Let me do the same thing and let it not plot every time.

Figures out a cost to go function, the optimal cost to go function, and an optimal policy. Now it looks a little noisy there. Again, we're going to talk about the sensitivity to discretization. But this is very much a bang bang policy, with the blue area being, do one action, the red area doing the other action.

The switching surface is actually pretty complicated. It's some complicated function of state, but it gets this beautifully smooth cost to go function, OK. Now let's take a second and look at the phase plots here. Let me actually do it in order here.

So this is the phase plot of the damped passive pendulum, OK, the original one we thought about in class. I just drew a few lines to help you. So if I start at downright position with a little bit of velocity, I'd slow down and stop. If I start near an unstable fixed point with near 0 velocity, then I actually fall down and go like this and end up standing still near the closest unstable fixed point, OK.

Now if I do my feedback linearization invert gravity controller to stabilize the fixed point, then what's the phase plot going to look like? It's going to look just like this, but it's going to be moved over there, right? So let's make sure that's true.

Ah, what did I call it? Invert gravity. OK, yeah. So I see the exact same things. Used to be my stable fixed point are now going over to the closest unstable one. This works great. The only objection to it is it required an enormous amount of torque to just pretend like you're inverted gravity.

OK, so what's the minimum time solution going to look like?

AUDIENCE: It's going to depend on what your torque constraint is.

PROFESSOR: It's going to depend on my torque constraint is, yeah. So for whatever torque constraint I have now, you could even figure out the units here. My torque constraint was chosen to be something like half of the stall torque required to hold out like this. Then let's see what happens.

This is the minimum time solution, which is exactly right. If I had more torque to give, it could have gotten out there quicker. And this added enough that, after going around once, it could get up to the top, OK.

Let me see. Why is it not drawing anymore? I've got this [INAUDIBLE].

Oop. So that was-- that's a random initial condition. So from the one I had shown, it took one pump. That one took two pumps, and that gets it to the top.

OK, but now, remember, my original challenge was to not just get to the top in minimum time. This is minimum time with bounded torque, so that's a little bit more satisfying. I don't want to pump in more torque than I could possibly implement. But what if I want to be sensitive about the torque? I want to get to the top, but I don't want to use a bunch of energy.

OK, now the quadratic cost function makes a lot of sense, OK. So I'm going to put a quadratic cost on being away from the top and a big quadratic cost on using actions. So that'll give me some sense of minimally stabilizing the top, OK. What's that one going to look like? Would you expect it to look like-- phase plot going.

AUDIENCE: Basically in phase space, it will more turns to get up there on the top [INAUDIBLE].

PROFESSOR: OK. What about if it's near the top? Is there going to look like a damp pendulum at the top? What's it going to do?

AUDIENCE: Well, if it's headed the wrong way near the top, it will probably swing all the way around.

PROFESSOR: Good. Right.

AUDIENCE: But if you put too much cost on distance, it might end up quickest on the [INAUDIBLE].

PROFESSOR: Perfect, OK. So let's switch this to be my quadratic regulator cost. Right, so that's what you said. Took more pumps to get up. And if you plot the phase plot from a couple of these different places-- oh. Crap, sorry. I thought I picked initial conditions that were far enough to show you that.

This is what you said, OK. This one happens to be close enough that it got to the top. This one took a lot of pumps and got out there.

But the point I was trying to illustrate-- I guess I need to either penalize torque a little bit more or-- I never change things by a factor of 2. It's too slow. Oh, I made it not move.

[LAUGHTER]

Sorry. But it showed my point, OK. So yeah, it has no incentive to move from the bottom. It says, I'm going to incur more cost by moving than by getting close to the goal. Not getting close. OK, but up at the top, it is able to-- given it was near the top with some velocity, with a little effort, it's worth going around and stabilizing itself at the top. Yeah? OK. Good.

AUDIENCE: If you iterate it far enough, it should go at the top, but--

PROFESSOR: No. Let's see. So--

AUDIENCE: It's because of the damping.

PROFESSOR: It's because of the damping.

AUDIENCE: Oh, OK.

PROFESSOR: Yeah, good. Because that is actually the steady state solution I'm plotting.

AUDIENCE: Oh.

PROFESSOR: Mm-hmm. OK.

[RUSTLING]

So if you care about simple pendula-- sorry-- and you want optimal solutions, this looks like a pretty satisfying way to do it. You could up with your arbitrary cost functions and see what you get. It runs in no time on my laptop, and you get things that look like optimal policies, nice phase plots, you name it, OK.

What's the catch? First catch is, how do I do the interpolation? How do I make that transition matrix? So on my pendulum example, I discretized some states. I have a handful-- I've already discretized actions, and I've got some other states over here that they've already discretized.

I'd have to be pretty remarkably lucky to have it that the random actions that I chose, integrated for some small amount of time, actually landed right on top of one of my other states. In fact, they tend to land in between the states, OK, so we do a little bit of interpolation between them. And one of the reasons I showed you that transition matrix form is that it's actually quite OK, quite standard, to say that my transition matrix, my T from S to S prime as a function of a is some-- let me just handwave it here-- but is some interpolated set of weights for S1 close.

[LAUGHS] OK. Zach just showed me a sign that said, the pendulum works. Having Matlab licensing issues. So we might-- I was hoping to run these on the real pendulum. We'll do it on Tuesday if not today. [LAUGHS] I don't know whey he didn't just say that, but there's a big bright green sign.

So let me write it like this for the moment, OK. So if I end up being near some states in two dimensions, I tend to interpolate between the three closest states, OK. So I'll call those Sy and S1-- Si, Sj, Sk, and I get some interpolants, W1, W2, and W3. They'd better sum to one, OK. And there's actually lots of ways to do that.

So actually, in previous times I've given the class, I went into some detail about that. I think that you could-- if you care about it, there's a lot of ways to do that. You could use the Matlab interp2 function. The one we use is called barycentric interpolation.

In the RL community, that was popularized by Munoz and Moore. That'll be cited in the notes. And it uses-- if you're operating in an N-dimensional space, it uses N plus 1 interpolants.

So in a two-dimensional space, it uses the three closest points. If you're in a four-dimensional space, it uses the five closest points, OK. And there's a very clean, simple algorithm to find the factors of that interpolant, OK.

The caveat is that everything spreads out. If I simulate my dynamics, my graph dynamics, what it's roughly saying is that if I started from this state, I'm going to be a little bit in that state, a little bit in that-- a little bit in state 48, a little bit in state 52. And then my transition's out of there, so I get this diffusion across my graph of where my state is, if that makes sense. Yeah? And that's why you get some of the smoothing effects that you saw in the plots, OK.

There's a bigger problem with that. The smoothing effects a lot of times don't look too dangerous, but they can do bad things to your solution if you're not careful, OK. So the big caveat is the solution you get is optimal only for the discrete system. We hope that it's approximately optimal for continuous, but compared to the finite element analysis world or the computational fluid dynamics world, or other people that solve these kind of problems, we have relatively less strict understanding of when-- of how bad this approximation can be compared-- based on the discretization.

There might actually be people out there that know it. I don't know how to tell you how bad it's going to get with the appro-- with the discretization. But I will ask you on your problem set to plot the bang bang solution of the double pendulum-- or, sorry, of the double integrator, and plot the analytical solution on top of it. And you'll see that if you're not careful, it's not just a little bit wrong. It can be systematically wrong. The switching surface turns out in the wrong place. And we'll ask you to think a little bit about why that is, OK.

That's really the only caveat if you about low dimensional problems. The more cited one, though, of course, is that there's this curse of dimensionality. The only reason that everybody doesn't use this stuff is because if I had a 10 degree of freedom robot and I had to break up that 10-dimensional space in discrete points, discrete buckets, and made a graph, I would need a bigger computer. Not just a little bit bigger, an exponentially bigger computer, OK. So you have to be able to discretize your space, and discretizing the space is exponentially expensive in the number of states, OK.

But so people actually-- historically, value methods were very popular in the '80s, say. And there's a lot of work that we're going to talk about that continues to be popular, about using approximations, where you don't do a strict discretization, but you do it so to try to approximate these costs, these dynamic programming algorithms with function approximation. But because of this sort of curse of dimensionality, a lot of people switched gears to a different class of optimization algorithms based more on the Pontryagin principle and more on gradient methods. We're going to talk about those, too.

But I think we have to remember that since the 1980s, our computers actually got a lot better, OK. Sounds silly, but so in the '80s, they could tile two-dimensional spaces, and three-dimensional hurt. Now we could probably do four, five, six-dimensional spaces, OK.

We actually did for-- we made that airplane land on a perch by just tiling the state space and doing brute force computation on that, OK. So you should look around. If there's some hard control problems that are four-dimensional or less that you consider to be unsolved, you could probably just hand them the dynamic programming and get a very nice solution, OK. And say, hey, you couldn't do it 10 years ago, but I can do it today on my laptop.

Awesome. OK. So unless Zach appears here, there's only one last thing I want to say, and that is I want to observe quickly-- we talked about the fact that optimal policies are not unique. But there's more things you can learn by staring at these guys a little bit. Let's put my R down to something more manageable.

Go, go, go. OK. Can you see it in this? It's a little bit hard to see it. I think you can see it if I turn the lights down. This is the quadratic regulator again. Now this isn't quite the quadratic regulator from the double integrator. This is now a quadratic cost function on a nonlinear dynamical system, OK.

In this case, the dynamics are smooth. They're non-linear, but they're smooth. There's nothing that changes abruptly in the derivatives. And the cost function is smooth, but you can find that the optimal policy can actually still be discontinuous, OK.

So costs-- so why is it discontinuous? In this case, because if I'm here and I'm going this way, I want to push up, but at some point, I have to change my mind and go the opposite way to pump up energy and get to the top. So this pump up strategy is inherently discontinuous, OK.

So this is the Gordian knot of optimal control, is as soon as things stop being linear, computing optimal cost to go functions can get arbitrarily hard, OK. And that's why computation's so great, because it does that stuff for me. But know that it doesn't take much to make it so the cost to go function gets a lot more subtle. Mm-hmm.

Good. So the class will proceed taking these methods as far as we can, breaking them, and then showing you approximation methods that work in higher dimensional spaces. And when we give up on optimality all together, we'll do motion planning, and we're going to get to more and more interesting robots.

But this is really a key idea. So I hope that the intuition came through and-- through your problems set. And I can share some of this code and everything. I hope you play with it, and think about it, and change cost functions, and see what happens. OK, see you next week.