Planning in Continuous Spaces: Trajectory Optimization

Tom Silver

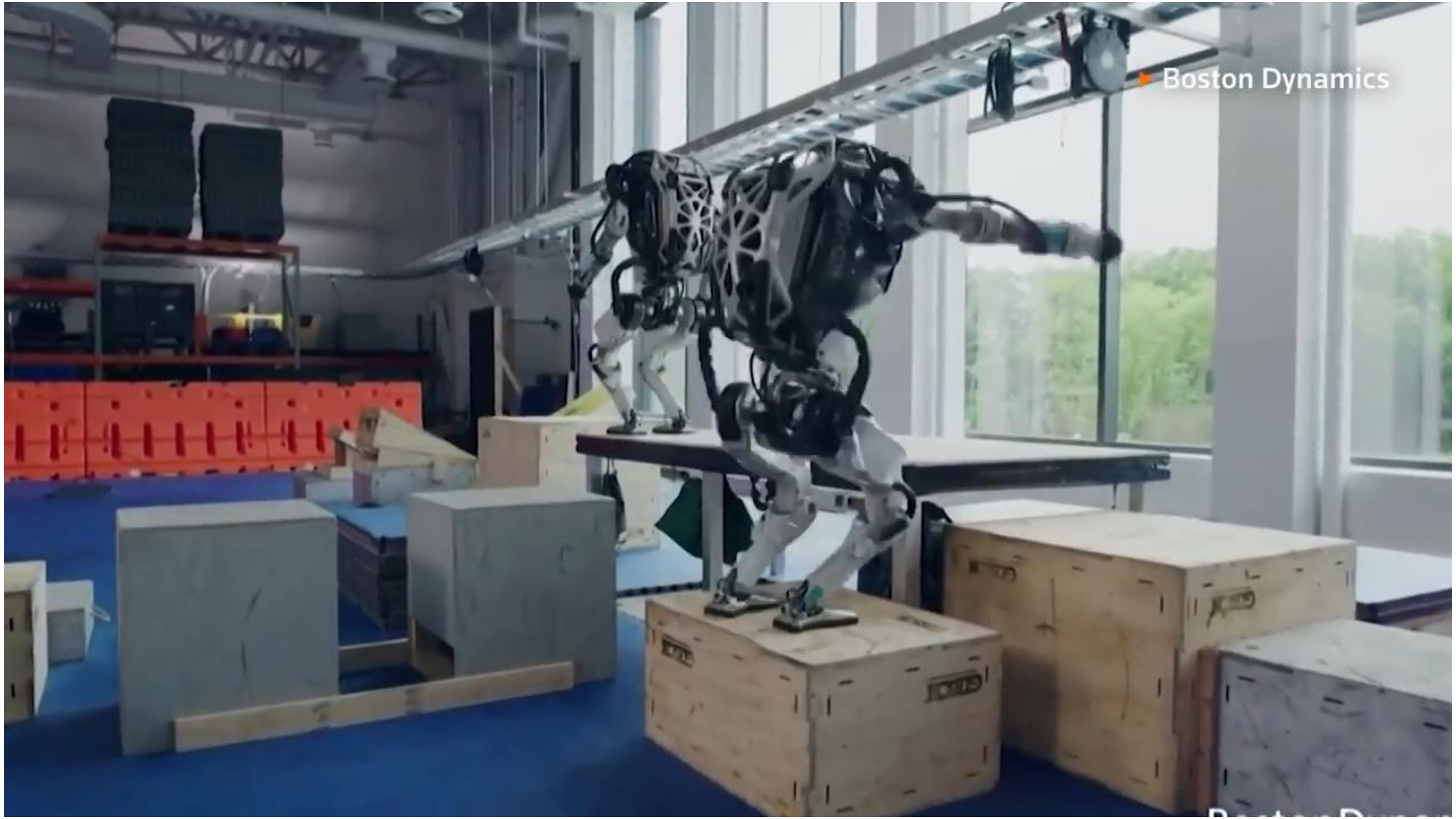
Robot Planning Meets Machine Learning

Princeton University

Fall 2025

Recap

- Last time: Motion planning, our first foray into continuous spaces
- This time: Planning in **more general continuous spaces**
- For example, planning in cases where dynamics are important
- Continue assuming full observability and determinism
- New material and then full-course review game!


Boston Dynamics

Trajectory Optimization Problems

We will consider *discrete-time*, *finite-horizon*, *deterministic* problems with:

1. A **state space** $\mathcal{X} \subseteq \mathbb{R}^n$
2. An **action space** $\mathcal{U} \subseteq \mathbb{R}^m$
3. A **transition function** $F: \mathcal{X} \times \mathcal{U} \rightarrow \mathcal{X}$
4. A **cost function** $C: (\mathcal{X} \times \mathcal{U})^* \times \mathcal{X} \rightarrow \mathbb{R}$
5. An **initial state** $x_0 \in \mathcal{X}$
6. A **time horizon** $H \in \mathbb{Z}^+$

Real-valued vectors

Over full trajectories! Common to sum over transition costs instead.

Trajectory Optimization Problems

Our objective is to find a **plan**

$$(u_0, u_1, \dots, u_{H-1})$$

with corresponding states

$$(x_0, x_1, \dots, x_H)$$

where $x_{t+1} = F(x_t, u_t)$

that **minimizes**

$$C(x_0, u_0, x_1, u_1, \dots, u_{H-1}, x_H)$$

Example: Double Integrator

State space: $\mathcal{X} \subseteq \mathbb{R}^2$

Position and Velocity

Action space: $\mathcal{U} \subseteq \mathbb{R}$

Acceleration

Transition function:

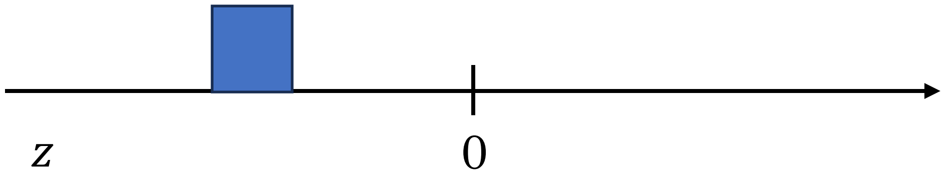
$$F\left(\begin{bmatrix} z_t \\ \dot{z}_t \end{bmatrix}, u_t\right) = \begin{bmatrix} z_t + \dot{z}_{t+1}\Delta t \\ \dot{z}_{t+1} + u_t \Delta t \end{bmatrix}$$

Δt = 0.1

Cost function:

$$C(\dots) = \sum_t z_t^2 + \ 0.1\dot{z}_t{}^2 + 0.01u_t^2$$

Initial state: $\begin{bmatrix} -1 \\ 0 \end{bmatrix}$ **Horizon:** 25



Example: Inverted Pendulum

State space: $\mathcal{X} \subseteq \mathbb{R}^2$

Angle and Velocity

Action space: $\mathcal{U} \subseteq \mathbb{R}$

Torque

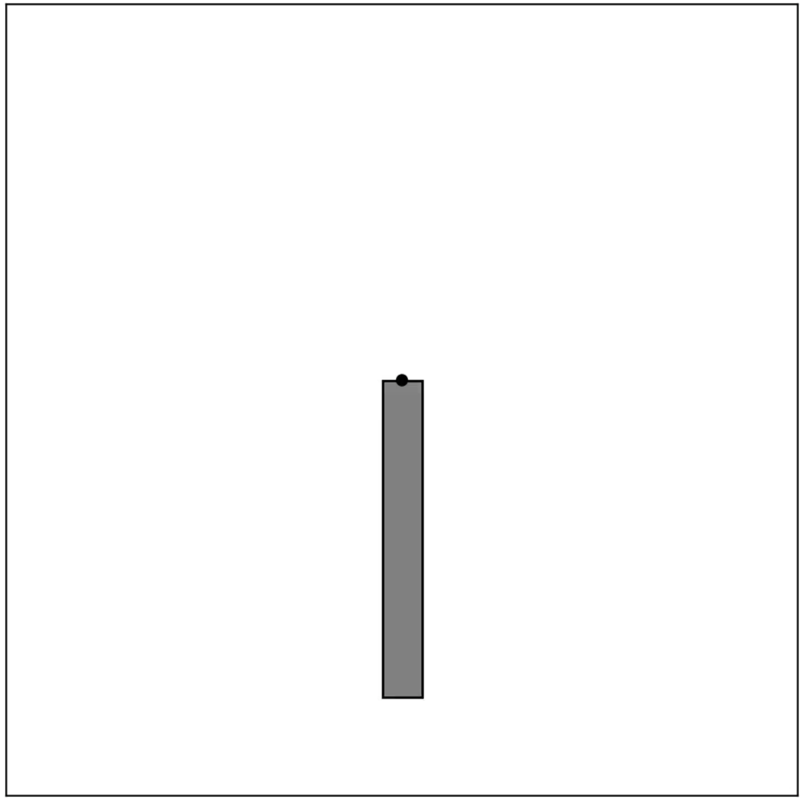
Transition function:

$$F\left(\begin{bmatrix}\theta_t \\ \dot{\theta}_t\end{bmatrix}, u_t\right) = \begin{bmatrix}\theta_t + \dot{\theta}_{t+1}\Delta t \\ \dot{\theta}_t + (k_1 \sin(\theta_t) + k_2 u_t)\Delta t\end{bmatrix}$$

Cost function:

$$C(\dots) = \sum_t \theta_t^2 + \ 0.1\dot{\theta}_t^2 + 0.01u_t^2$$

Initial state: $\begin{bmatrix}\pi \\ 1\end{bmatrix}$ **Horizon:** 200



Stupidest Possible Algorithm

Repeat until impatient:

1. Sample a plan $(u_0, u_1, \dots, u_{H-1}) \sim \boldsymbol{U}$
2. Run through F to get $(x_0, x_1, \dots, x_H)$
3. Evaluate $C(x_0, u_0, x_1, u_1, \dots, u_{H-1}, x_H)$

For example, from some Gaussian distribution

Return the best seen plan

Trajectory Optimization Taxonomy

What is optimized?	
Actions *a.k.a. indirect or shooting*	States and Actions *a.k.a. direct transcription or collocation*

What derivatives (of dynamics and costs) are required?		
Zero-order	First-order	Second-order

To what extent is the approach specific to *trajectory* optimization?		
Not really at all	Some	A lot

Stupidest Possible Algorithm

What is optimized?	
Actions *a.k.a. indirect or shooting*	States and Actions *a.k.a. direct transcription or collocation*

What derivatives (of dynamics and costs) are required?		
Zero-order	First-order	Second-order

To what extent is the approach specific to *trajectory* optimization?		
Not really at all	Some	A lot

Can We Do Better in this Category?

What is optimized?	
Actions *a.k.a. indirect or shooting*	States and Actions *a.k.a. direct transcription or collocation*

What derivatives (of dynamics and costs) are required?		
Zero-order	First-order	Second-order

To what extent is the approach specific to *trajectory* optimization?		
Not really at all	Some	A lot

Shooting as Unconstrained Optimization

Let $\boldsymbol{u} = (u_0, u_1, \ldots, u_{H-1})$ and let $C(\boldsymbol{u})$ be the corresponding cost

Restating our objective:

$$\min_{\boldsymbol{u}} C(\boldsymbol{u})$$

This is now just an unconstrained continuous optimization problem

Let's leverage tools…

Zero-Order Continuous Optimization

A.k.a. “derivative-free” or “blackbox” optimization

Notable examples:

- **Random search**
- **Cross-entropy method**
- Bayesian optimization
- Evolution strategies
- Nelder-Mead

We will briefly discuss these

Random Search

```
Initialize $\boldsymbol{u}$ randomly
Repeat:
    Sample $\boldsymbol{u}$' in the neighborhood of $\boldsymbol{u}$
    If $C(\boldsymbol{u}') < C(\boldsymbol{u})$:
        $\boldsymbol{u} \leftarrow \boldsymbol{u}'$
```

Limitation: sampling does not adapt!

When might this work better than our SPA?

Many possible variations on “neighborhood”

> “Algorithms that are invented independently by four different communities probably have something good going for them.” – Ben Recht (2018)

Cross-Entropy Method

Define $\mathrm{P}(\boldsymbol{u} \mid \boldsymbol{\theta})$ for some initial $\boldsymbol{\theta}$

Repeat:

Example: Gaussian; $\boldsymbol{\theta}$ is mean and variance

Sample N times from $\mathrm{P}(\boldsymbol{u} \mid \boldsymbol{\theta})$

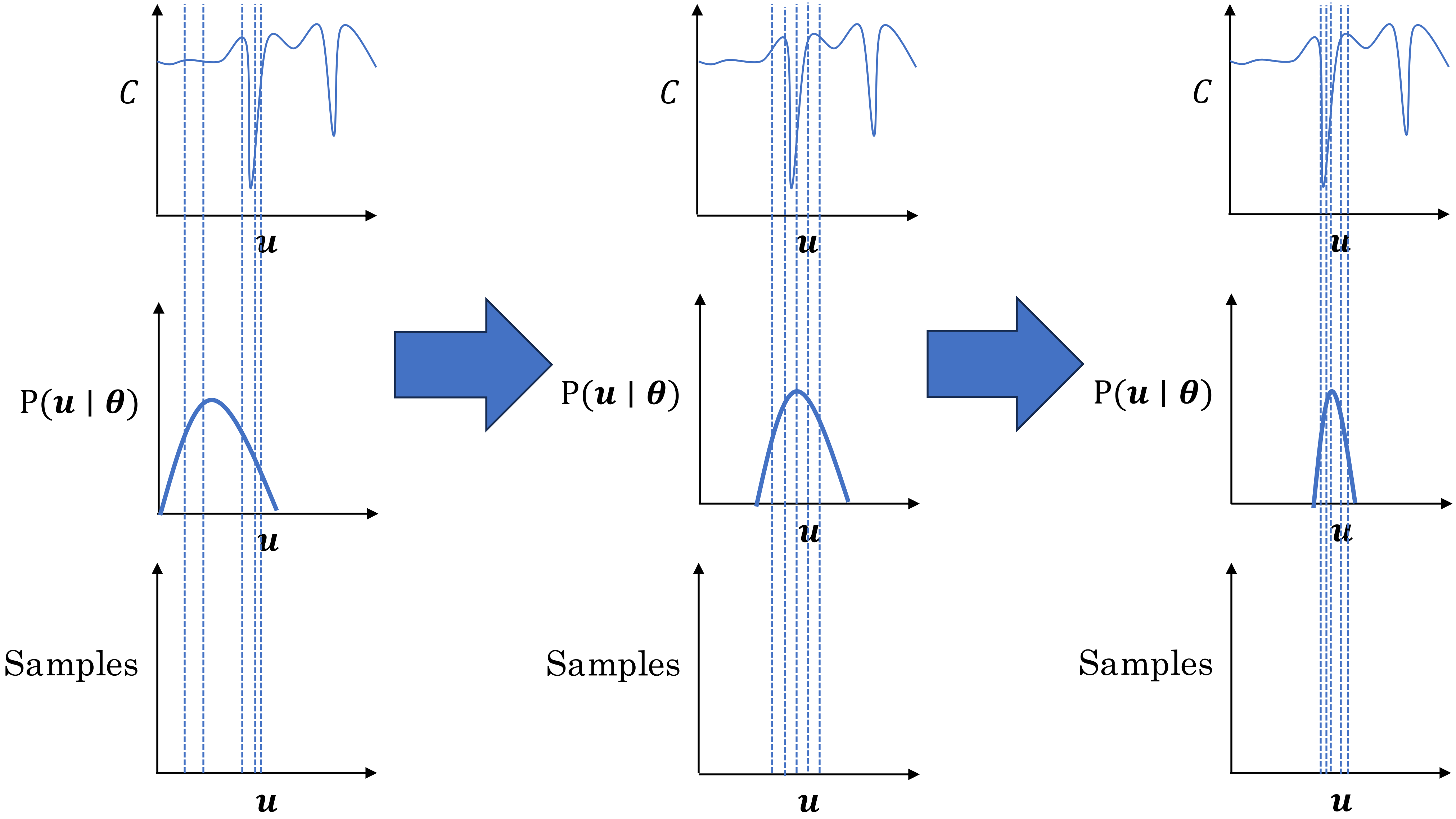
Order the samples by cost: $C(\boldsymbol{u_1}) < C(\boldsymbol{u_2}) < \ldots < \boldsymbol{C}(\boldsymbol{u_N})$

Keep the top K samples: $[\boldsymbol{u_1}, \ldots, \boldsymbol{u_K}]$

Fit a new distribution to the samples: $\boldsymbol{\theta} = \boldsymbol{fit}([\boldsymbol{u_1}, \ldots, \boldsymbol{u_K}])$

Example: compute mean and variance

Illustration of CEM



General Trick 1: Model-Predictive Control

Given problem with initial state x_0 and horizon T

Repeat:

1. Solve for $(u_0, u_1, \ldots, u_{H-1})$
2. **Execute** the first action u_0
3. Update x_0 to the new state
4. Update $T = T - 1$

Can also look T' < T steps ahead: “receding horizon control”

Common to run *one step* of iterative optimization

Important: warm-start optimization from previous step

Main benefit: solving can be very approximate, as long as it’s fast

General Trick 2: Optimize Splines Instead

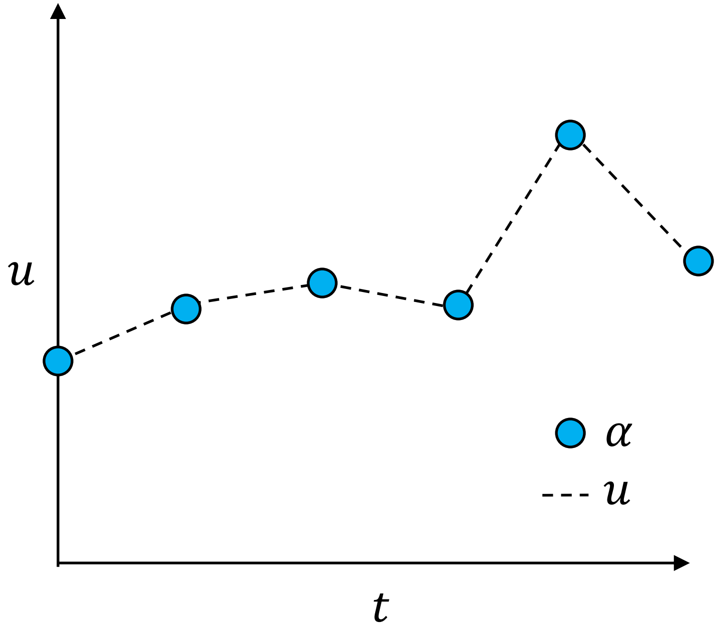
Optimizing $(u_0, u_1, \ldots, u_{H-1})$ is slow for large H

Instead, optimize over lower-dimensional α:

$$u_t = f(t, \boldsymbol{\alpha}) \text{ where } \boldsymbol{\alpha} \in \mathbb{R}^d \text{ and } d \ll mH$$

Common: think of α as "action waypoints" and interpolate between them

For example, linear splines (see right)



General Trick 3: Initialize Well

For iterative methods (which most are), the initialization matters!

Extreme case: initialize at the global optimum

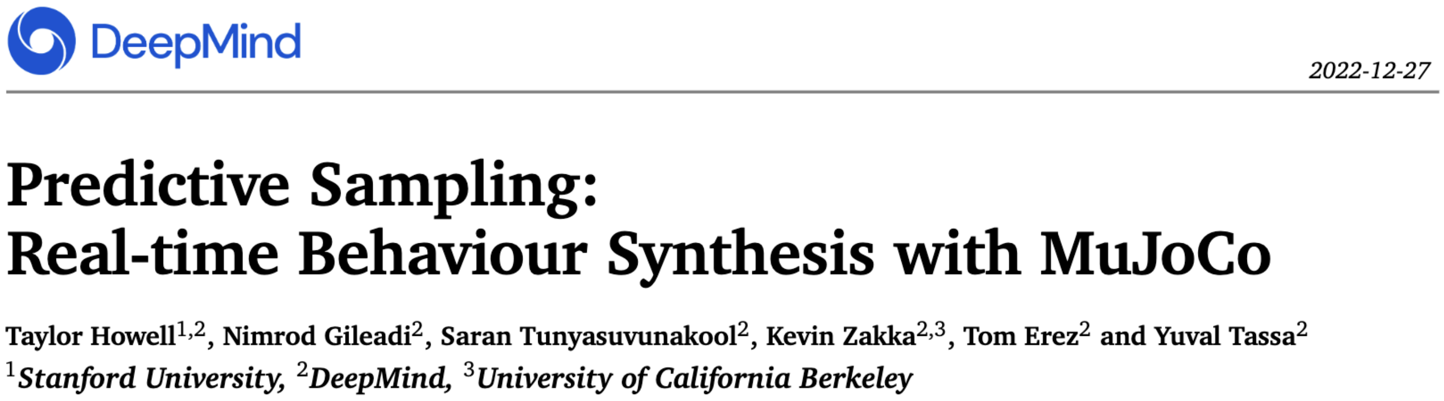
Also helpful for debugging

More common: try to initialize "near" a "good local optimum"

One trick: solve a *reduced* problem to get an initialization

- Similar in spirit to deriving heuristics from problem relaxations
- The backflipping BD robot does this!

Predictive Sampling



DeepMind

2022-12-27

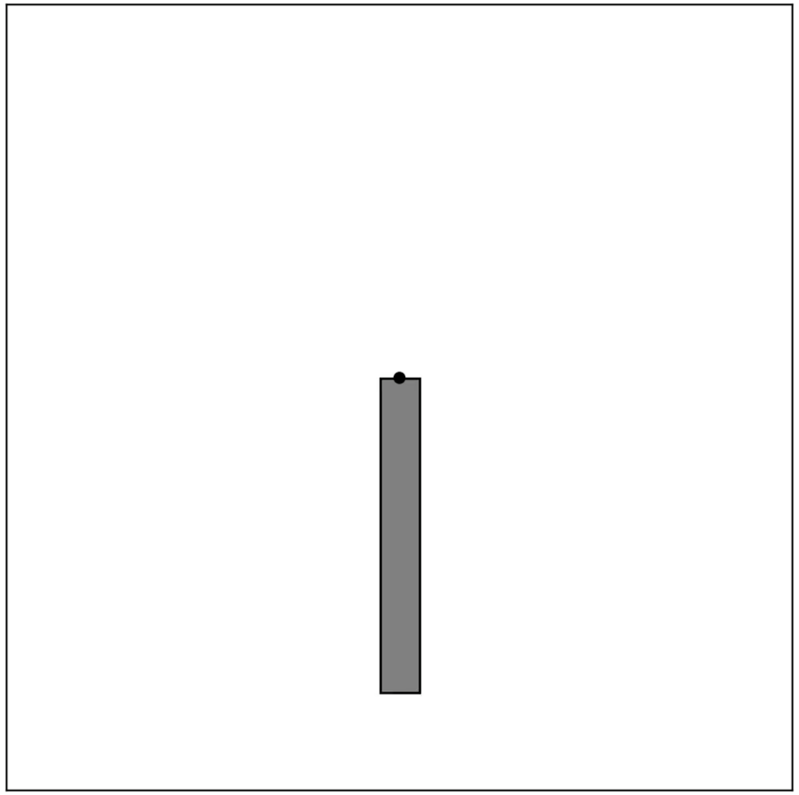
Predictive Sampling:
Real-time Behaviour Synthesis with MuJoCo

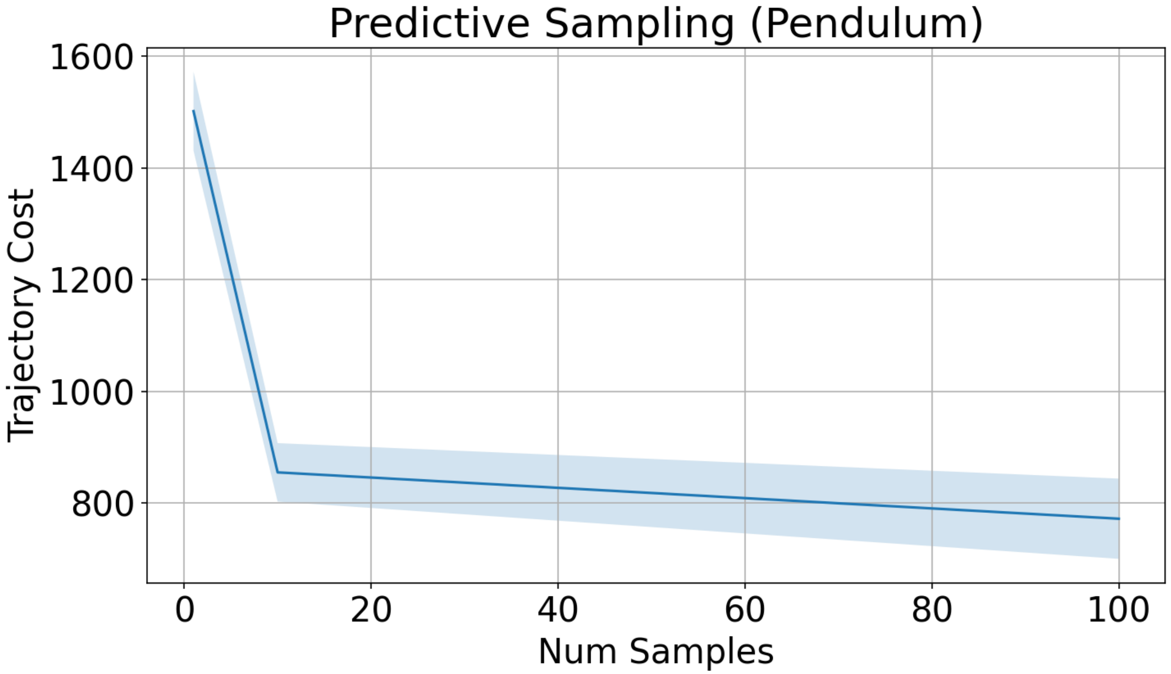
Taylor Howell[1,2], Nimrod Gileadi[2], Saran Tunyasuvunakool[2], Kevin Zakka[2,3], Tom Erez[2] and Yuval Tassa[2]
[1]*Stanford University,* [2]*DeepMind,* [3]*University of California Berkeley*

MPC + Splines + Random Search

Sometimes works surprisingly well

Predictive Sampling in Pendulum Env





Predictive Sampling

What is optimized?	
[✓ Actions *a.k.a. indirect or shooting*]	States and Actions *a.k.a. direct transcription or collocation*

What derivatives (of dynamics and costs) are required?		
[✓ Zero-order]	First-order	Second-order

To what extent is the approach specific to *trajectory* optimization?		
Not really at all	[✓ Some]	A lot

Gradient-Based Shooting Methods

What is optimized?	
[✓ Actions] *a.k.a. indirect or shooting*	States and Actions *a.k.a. direct transcription or collocation*

What derivatives (of dynamics and costs) are required?		
Zero-order	[✓ First-order]	Second-order

To what extent is the approach specific to *trajectory* optimization?		
Not really at all	[✓ Some]	A lot

Gradient Descent

Recall our objective: $\min_{\boldsymbol{u}} C(\boldsymbol{u})$

Suppose costs and dynamics are differentiable

Or approximate with finite differences (e.g., MuJoCo does this)

This sounds like a job for gradient descent!

Or SGD, or Adam, or whatever…

```
Repeat:
    $\boldsymbol{u} \leftarrow \boldsymbol{u} - \gamma \nabla C(\boldsymbol{u})$
```

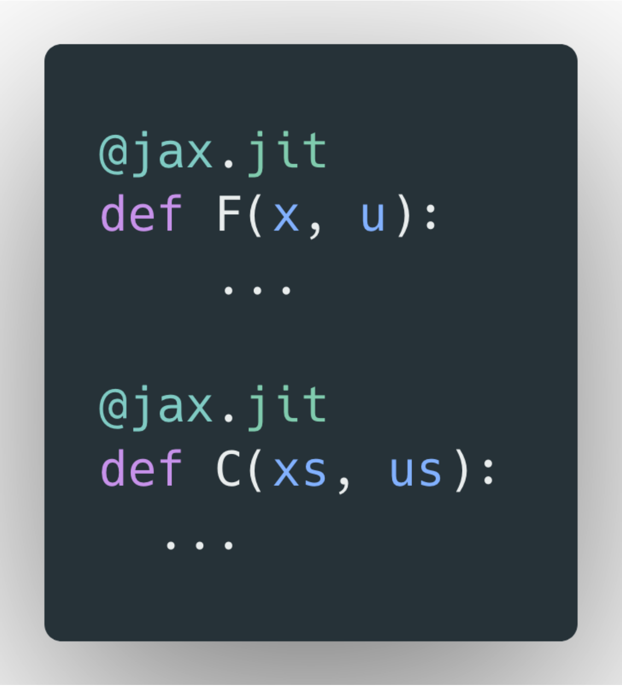
What is $\nabla C(\boldsymbol{u})$?

Calculate a Gradient? Myself?

Or we can let autodiff do it for us

Tensorflow, PyTorch, JAX, etc.

We then need to define the dynamics & costs in those terms



```
@jax.jit
def F(x, u):
    ...

@jax.jit
def C(xs, us):
  ...
```

Example: Double Integrator

State space: $\mathcal{X} \subseteq \mathbb{R}^2$

Position and Velocity

Action space: $\mathcal{U} \subseteq \mathbb{R}$

Acceleration

Transition function:

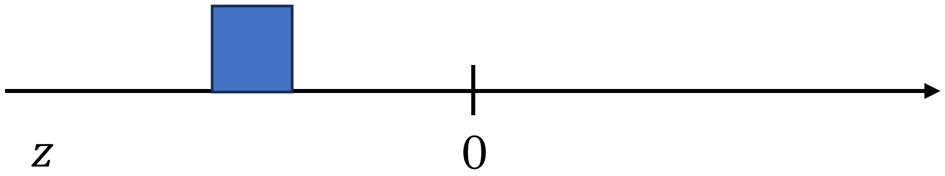
$$F\left(\begin{bmatrix} z_t \\ \dot{z}_t \end{bmatrix}, u_t\right) = \begin{bmatrix} z_t + \dot{z}_{t+1}\Delta t \\ \dot{z}_{t+1} + u_t \Delta t \end{bmatrix}$$

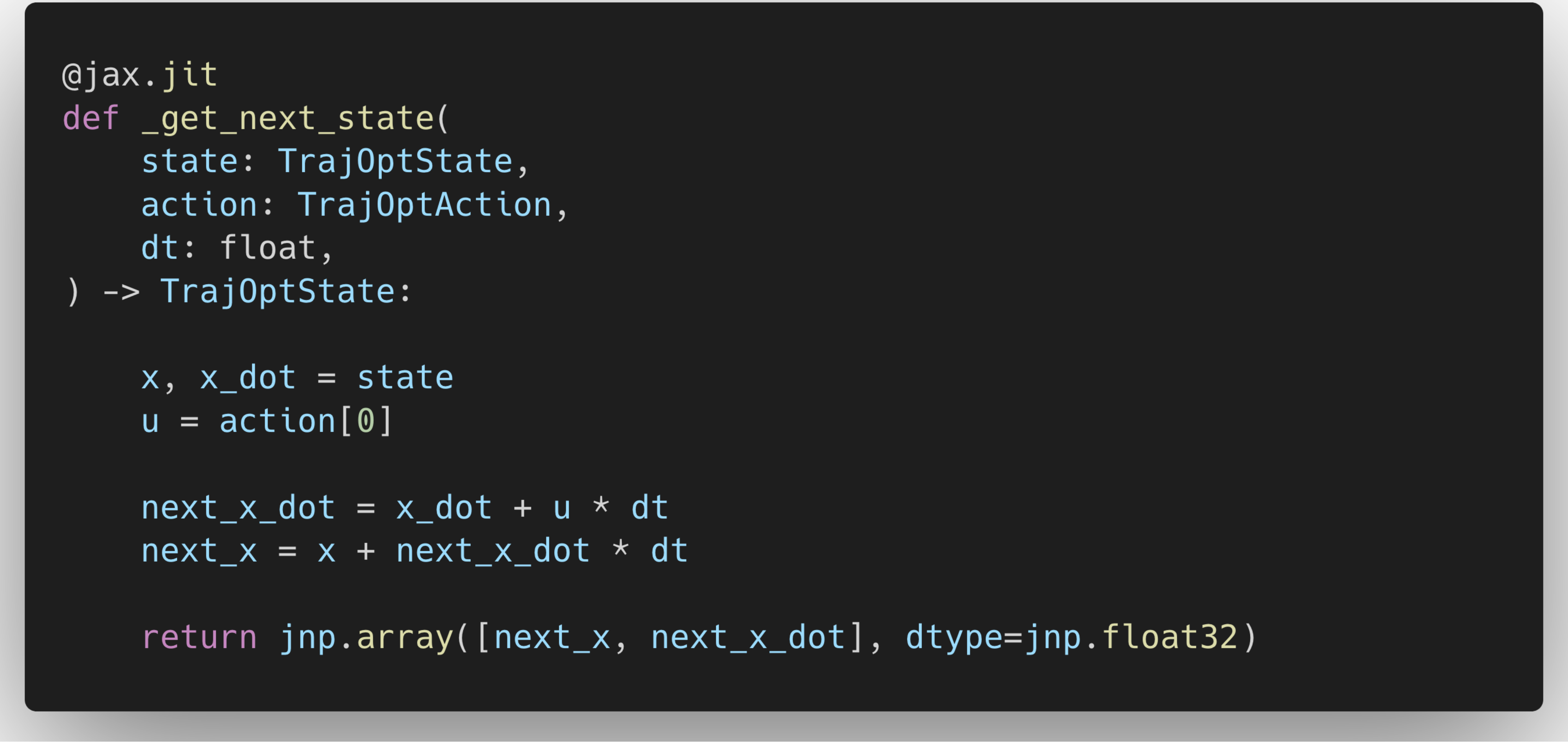
Δt = 0.1

Cost function:

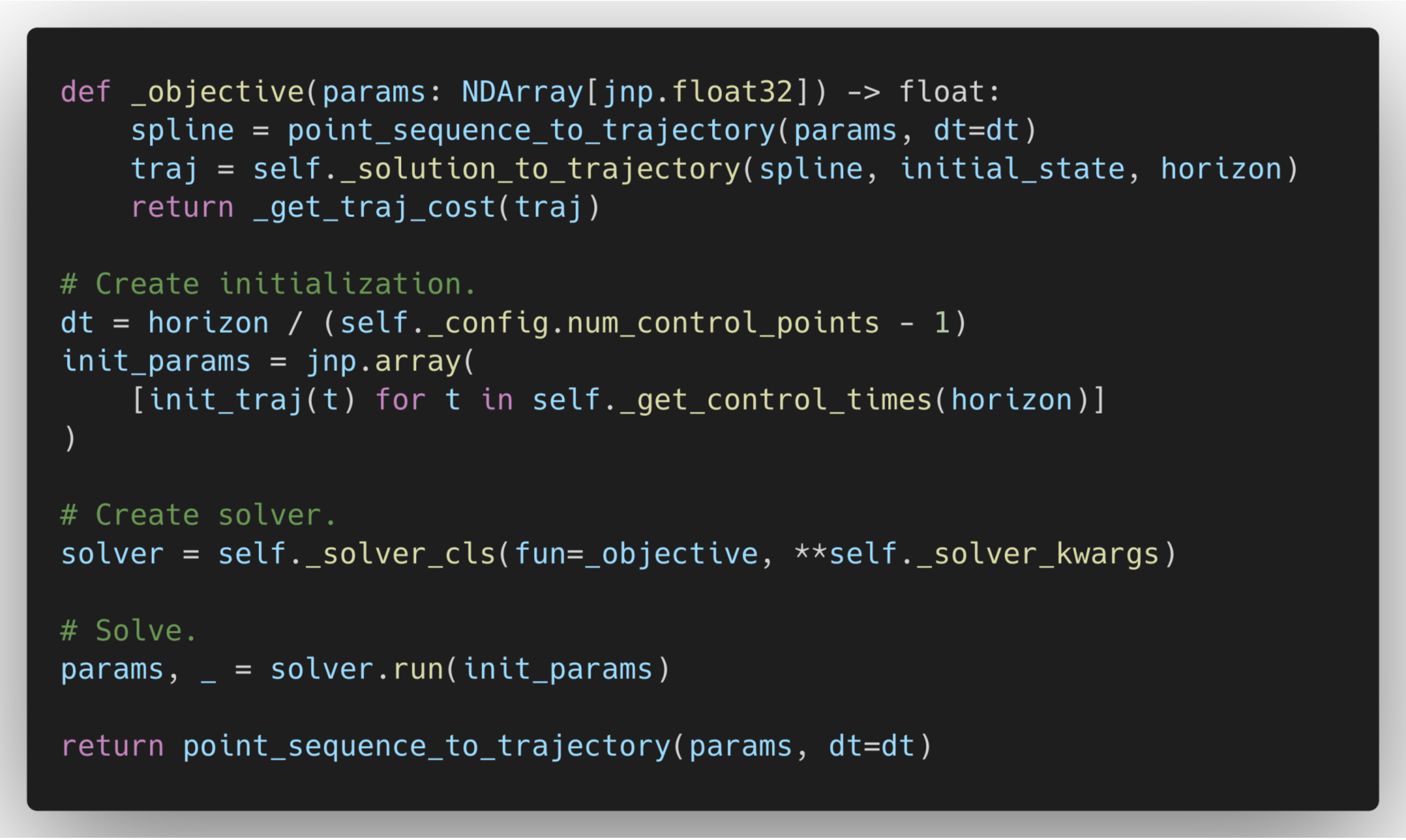
$$C(\ldots) = \sum_t z_t^2 + \ 0.1\dot{z_t}^2 + 0.01u_t^2$$

Initial state: $\begin{bmatrix} -1 \\ 0 \end{bmatrix}$ **Horizon:** 25



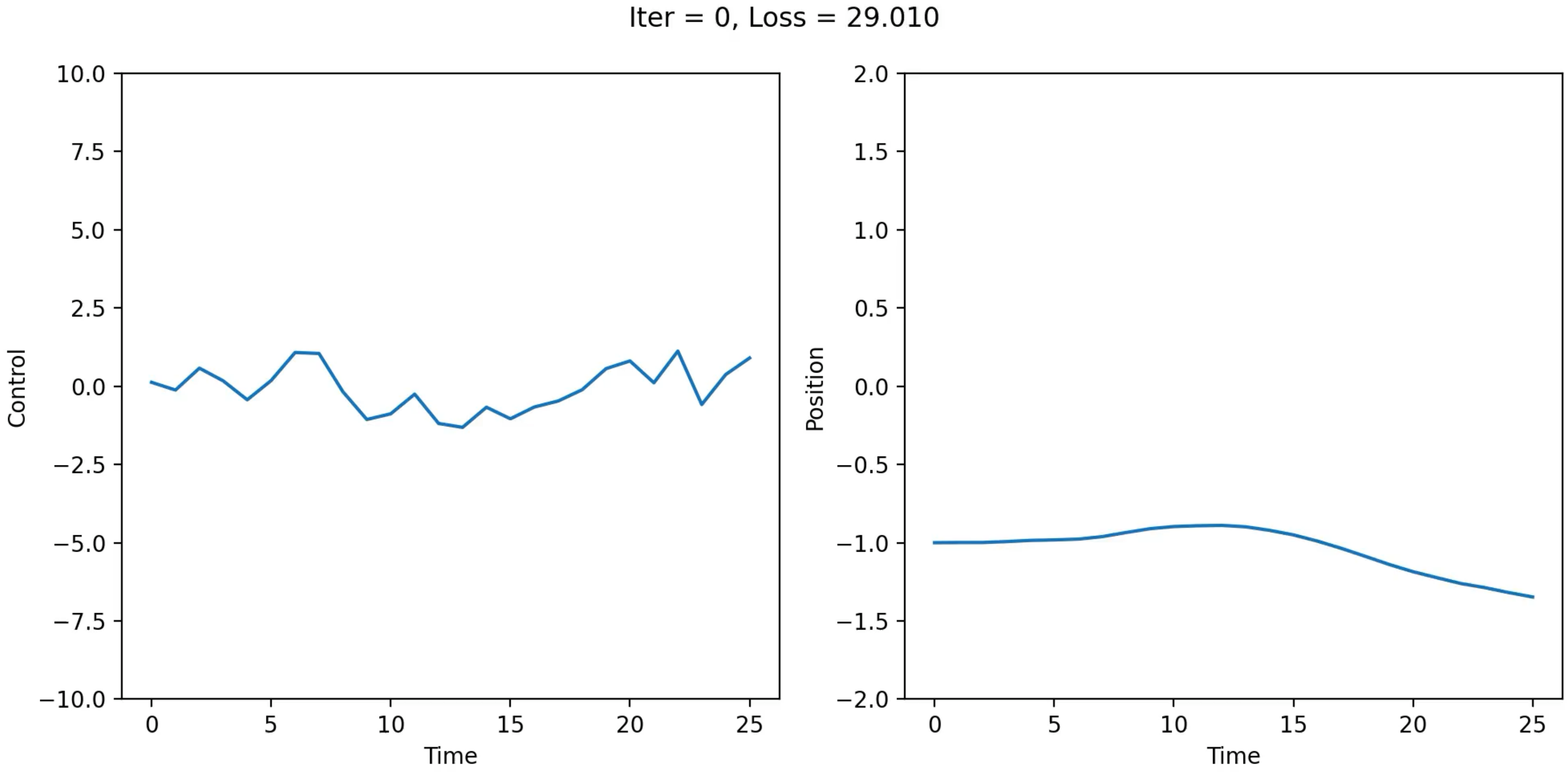


```python
@jax.jit
def _get_next_state(
    state: TrajOptState,
    action: TrajOptAction,
    dt: float,
) -> TrajOptState:

    x, x_dot = state
    u = action[0]

    next_x_dot = x_dot + u * dt
    next_x = x + next_x_dot * dt

    return jnp.array([next_x, next_x_dot], dtype=jnp.float32)
```



```
def _objective(params: NDArray[jnp.float32]) -> float:
    spline = point_sequence_to_trajectory(params, dt=dt)
    traj = self._solution_to_trajectory(spline, initial_state, horizon)
    return _get_traj_cost(traj)

# Create initialization.
dt = horizon / (self._config.num_control_points - 1)
init_params = jnp.array(
    [init_traj(t) for t in self._get_control_times(horizon)]
)

# Create solver.
solver = self._solver_cls(fun=_objective, **self._solver_kwargs)

# Solve.
params, _ = solver.run(init_params)

return point_sequence_to_trajectory(params, dt=dt)
```

Gradient Descent with JAX in Double Integrator



Behind the Scenes

JAX knows (in this problem) that $C(\boldsymbol{u}) = \sum_t c(x_t, u_t)$

$c(x_t, u_t)$ happens to be quadratic in this problem (not important)

and (in general) that

$$
\begin{aligned}
x_1 &= F(x_0, u_0) \\
x_2 &= F(F(x_0, u_0), u_1) \\
x_3 &= F(F(F(x_0, u_0), u_1), u_2)
\end{aligned}
$$

This is reminiscent of the function composition in neural networks…

Behind the Scenes

JAX is using reverse-mode autodiff (backpropagation)

1. Simulate **forward** to get $x_{t+1} = F(x_t, u_t)$
2. Calculate state gradients **backwards**:

“Co-state”

$$\lambda_{t-1} = \frac{\partial c(x_t, u_t)}{\partial x_t} + \frac{\partial F(x_t, u_t)}{\partial x_t}^T \lambda_t$$

“Adjoint equation”

Starting with $\lambda_H = \frac{\partial c(x_H)}{\partial x_H}$

Behind the Scenes

JAX is using reverse-mode autodiff (backpropagation)

1. Simulate **forward** to get $x_{t+1} = F(x_t, u_t)$
2. Calculate state gradients **backwards**
3. Calculate the action gradients

Take a moment to appreciate that this is way better than naïve gradient calculation!

$$\frac{\partial C}{\partial u_t} = \frac{\partial c(x_t, u_t)}{\partial u_t} + \frac{\partial F(x_t, u_t)}{\partial x_t}^T \lambda_t$$

Differentiating through Splines

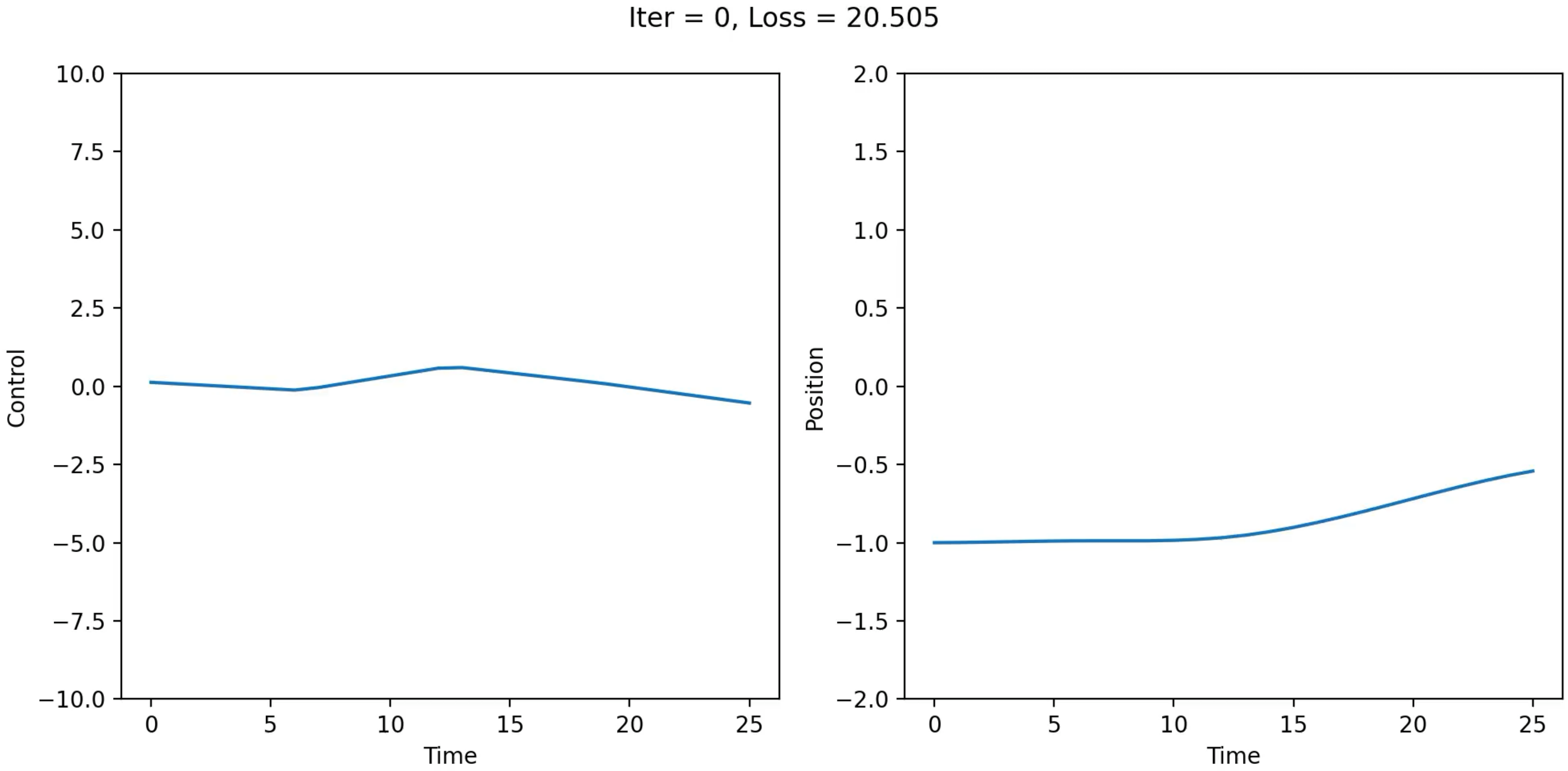
Recall spline trick: $u_t = f(t, \boldsymbol{\alpha})$

Chain rule again:

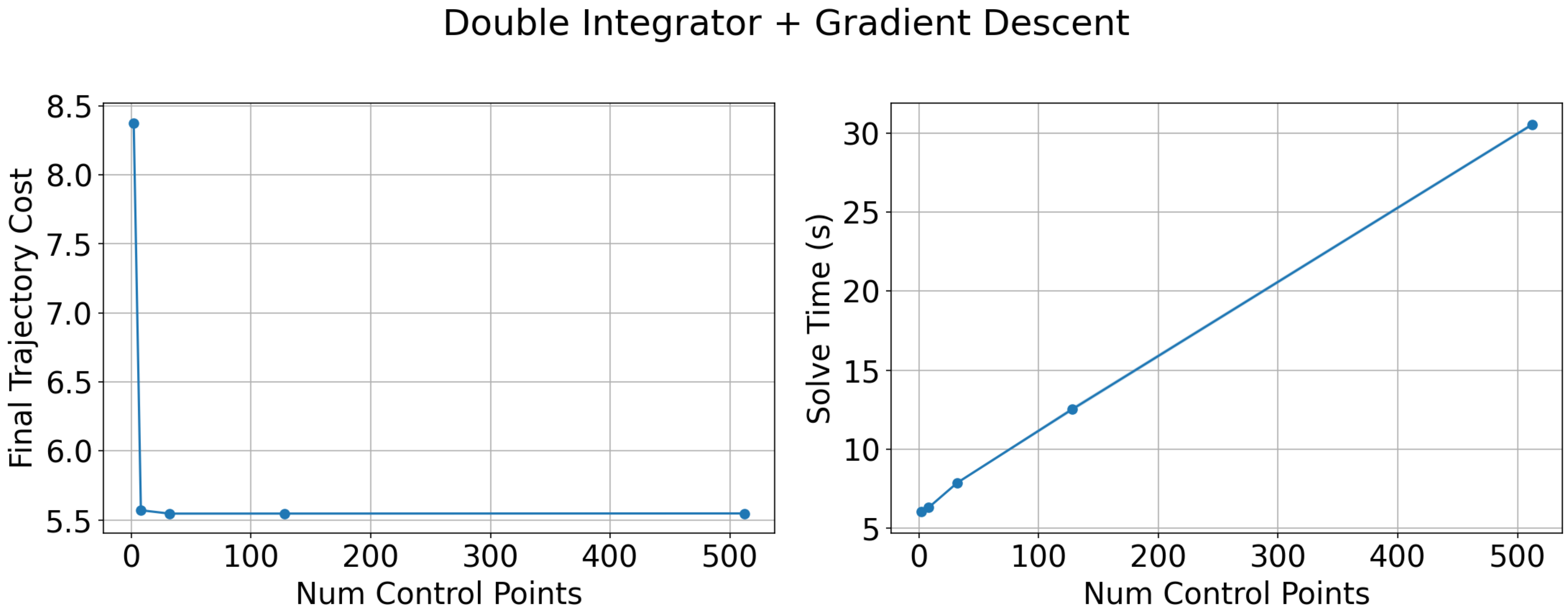
$$\frac{\partial C}{\partial \boldsymbol{\alpha}} = \frac{\partial C}{\partial \boldsymbol{u}} \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{\alpha}}$$

This is also handled easily by autodiff (so long as spline is implemented using autodiff)

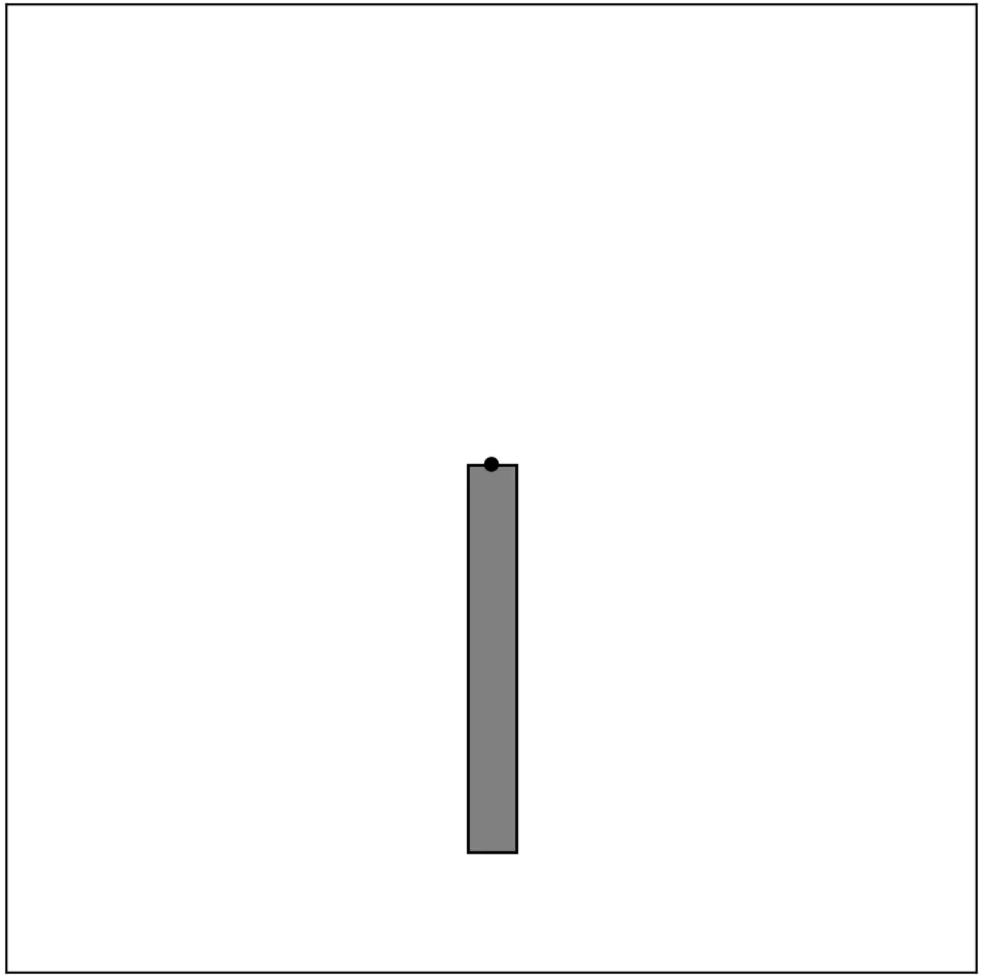
# control points = 5



Trading Off Speed and Cost



Gradient Descent in Pendulum



It is certainly possible to do better than this, but it is very finicky…

Differentiable Physics Engines

BRAX

Freeman et al. (2021)

See also:

- Dojo (Howell et al. 2022)
- End-to-End Differentiable Physics for Learning and Control (De Avila Belbute Peres et al. 2018)
- Tiny Differentiable Simulator (Coumans 2020)
- Several others

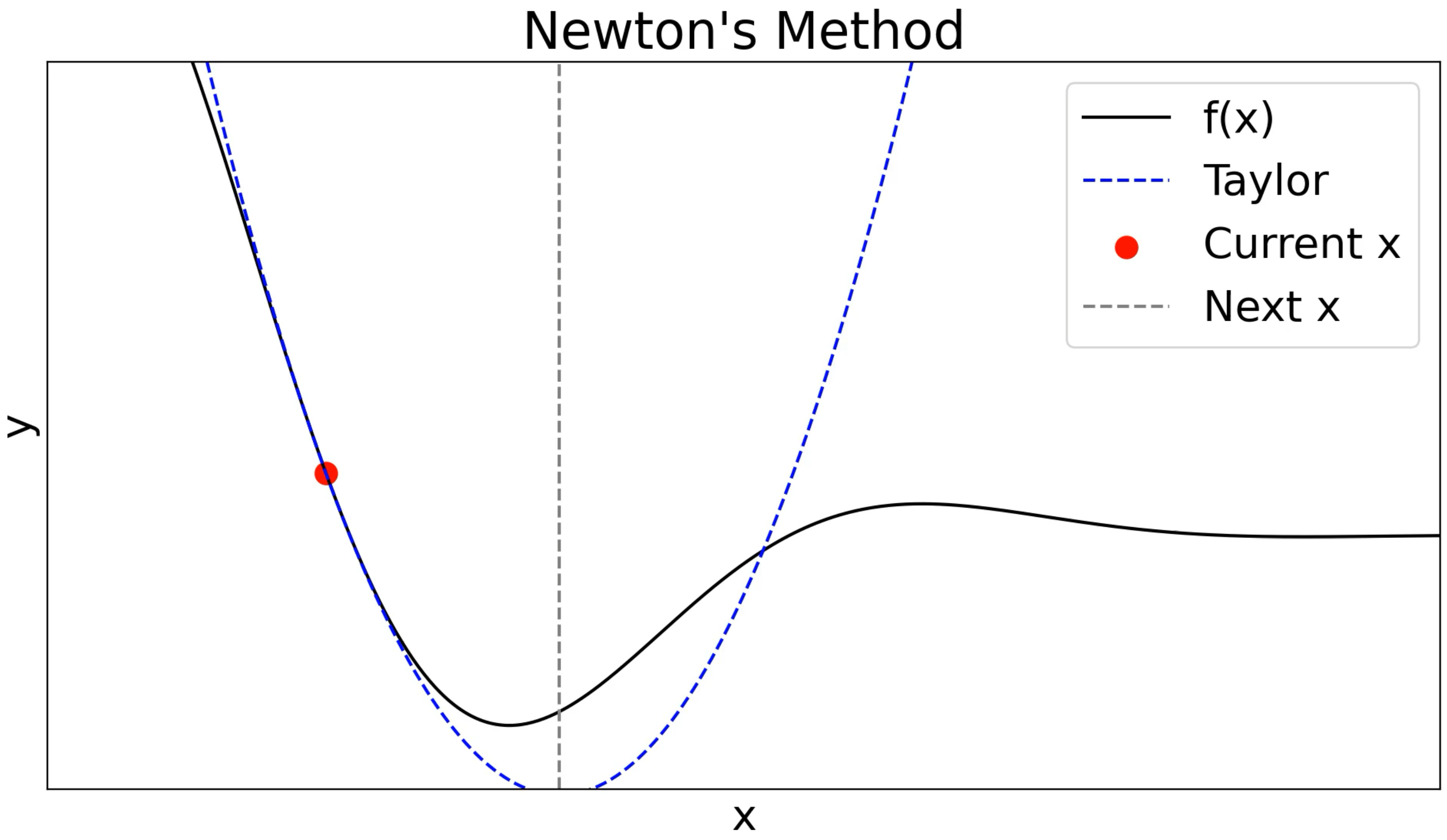
Second-Order Shooting Methods

What is optimized?	
Actions *a.k.a. indirect or shooting*	States and Actions *a.k.a. direct transcription or collocation*

What derivatives (of dynamics and costs) are required?		
Zero-order	First-order	Second-order

To what extent is the approach specific to *trajectory* optimization?		
Not really at all	Some	A lot

Newton’s Method (a.k.a. Newton–Raphson)



For a twice differentiable function $f(x)$ that we want to minimize:

$$x_{i+1} = x_i + [f''(x)]^{-1} f'(x)$$

(Initialize x_0 and repeat to converge)

Uses Hessian $f''(x)$, so 2nd order

Need to be careful if Hessian is not PSD; Taylor parabola would flip

Differential Dynamic Programming

What is optimized?	
[✓ Actions] *a.k.a. indirect or shooting*	States and Actions *a.k.a. direct transcription or collocation*

What derivatives (of dynamics and costs) are required?		
Zero-order	First-order	[✓ Second-order]

To what extent is the approach specific to *trajectory* optimization?		
Not really at all	Some	[✓ A lot]

Differential Dynamic Programming (DDP)

See excellent reference: “Control-Limited Differential Dynamic Programming” (Tassa et al. 2014)

Let’s apply Newton’s method to our shooting problem: $\min_{\boldsymbol{u}} C(\boldsymbol{u})$

As with gradient descent, want to avoid naïve computation

- Especially now that we need Hessians!

We will do a similar forward-backward procedure, but this time:

1. Estimate *cost-to-go (value function)* at each step
2. Use second-order info
3. Build up explicit function approximations

Differential Dynamic Programming (DDP)

(Hamilton-Jacobi-) Bellman Equations:

$$V_H(x_H) = c(x_H)$$
$$V_t(x_t) = \min_{u_t} c(x_t, u_t) + V_{t+1}(F(x_t, u_t))$$

Same as we're used to, just simpler because of no stochasticity

Forward pass, given nominal $\boldsymbol{u}$:

$$x_1 = F(x_0, u_0)$$
$$x_2 = F(x_1, u_1)$$
$$x_3 = F(x_2, u_2)$$
$$\ldots$$

Same as in gradient descent

Differential Dynamic Programming (DDP)

Consider the usual Q functions, but now centered around nominal trajectory. So the input is a *difference* with respect to the nominal:

$$Q_t(\partial x_t, \partial u_t) = c(x_t + \partial x_t, u_t + \partial u_t) + V_{t+1}\big(F(x_t + \partial x_t, u_t + \partial u_t)\big)$$

The 2nd order Taylor expansion of Q_t is:

$$Q_t(\partial x_t, \partial u_t) \approx \frac{1}{2}\begin{bmatrix} 1 \\ \partial x_t \\ \partial u_t \end{bmatrix}^T \begin{bmatrix} 0 & M_x^T & M_u^T \\ M_x & M_{xx} & M_{xu} \\ M_u & M_{ux} & M_{uu} \end{bmatrix} \begin{bmatrix} 1 \\ \partial x_t \\ \partial u_t \end{bmatrix}$$

Each submatrix M can be computed from step t+1 quantities!

Differential Dynamic Programming (DDP)

If $Q_t(\partial x_t, \partial u_t)$ is quadratic (which we are ensuring), then:

1. The optimal control modification is **linear** (affine):

$$\partial u_t^*(\partial x_t) = \mathrm{argmin}_{\partial u_t} Q_t(\partial x_t, \partial u_t) = k + K\partial x_t$$

2. The value function is **quadratic** (not shown)

This is what ensures the backward pass does not “blow up”!

Differential Dynamic Programming (DDP)

- Forward and backward pass can be repeated iteratively

- Updating nominal $\boldsymbol{u}$ after each iteration

- These iterations are Newton-Raphson steps

DDP Summary

1. It’s like dynamic programming from finite-horizon MDP land…
2. But instead of tabular value functions, we have quadratic ones
3. The quadratic functions are derived approximately from 2nd order Taylor expansions of the Q functions
4. So it’s really just Newton’s method + DP!
5. Remember: if the underlying system is nonlinear, this all could be terrible…

DDP → iLQR

Sometimes lower-order methods approximate higher-order ones

What derivatives (of dynamics and costs) are required?		
Zero-order	First-order	Second-order

Finite differences

Gauss-Newton (≈Newton)
→TrajOpt: **iLQR** (≈DDP)

iLQR → LQR

If the underlying dynamics are linear and costs are quadratic, then one step of iLQR is enough to get optimal performance

Similarly: Newton's method finds global optimum in 1 step if function is quadratic

This is called **LQR (Linear Quadratic Regulator)**

Bonus: adding Gaussian noise doesn't really change anything (except the name: LQG)

Very well-studied system in control theory

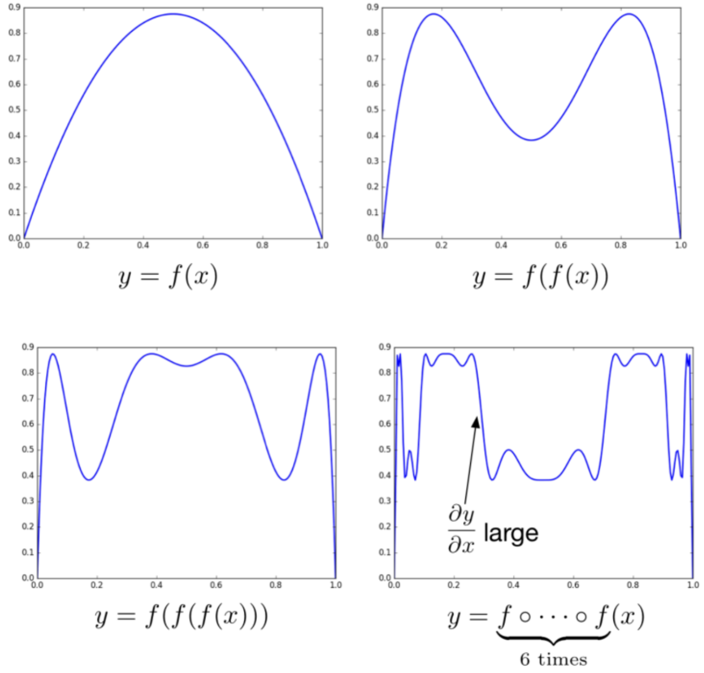
Beware of Optimizing Iterated Functions

Shooting requires repeatedly composing:

$$x_1 = F(x_0, u_0)$$
$$x_2 = F(F(x_0, u_0), u_1)$$
$$x_3 = F(F(F(x_0, u_0), u_1), u_2)$$

…

This can lead to very difficult optimization landscapes; issues like RNN training



$y = f(x)$

$y = f(f(x))$

$y = f(f(f(x)))$

$y = \underbrace{f \circ \cdots \circ f}_{6 \text{ times}}(x)$

Figure 2: Iterations of the function $f(x) = 3.5\,x\,(1 - x)$.

Figure from Roger Grosse (2017)

A Different Way….

What is optimized?	
Actions *a.k.a. indirect or shooting*	States and Actions *a.k.a. direct transcription or collocation*

What derivatives (of dynamics and costs) are required?		
Zero-order	First-order	Second-order

To what extent is the approach specific to *trajectory* optimization?		
Not really at all	Some	A lot

Direct Transcription

$$\min_{\boldsymbol{x},\boldsymbol{u}} \quad C(\boldsymbol{x},\boldsymbol{u})$$
$$\text{subject to} \quad x_{t+1} = F(x_t, u_t)$$

Optimizing states too!

Equality **constraints**

At first glance…

- Aren't we just making the optimization harder?
- Now the optimizer needs to "figure out" the dynamics, too…

But maybe…

- We are giving the optimizer more freedom
- For example: it could find good future states and then "reason backwards" to get actions

Also: since we're now doing *constrained optimization*, we could easily incorporate other constraints

Constrained Optimization

$$\begin{aligned} \min_{z} \quad & f(z) \\ \text{subject to} \quad & g_i(z) = 0 \\ & h_j(z) \geq 0 \end{aligned}$$

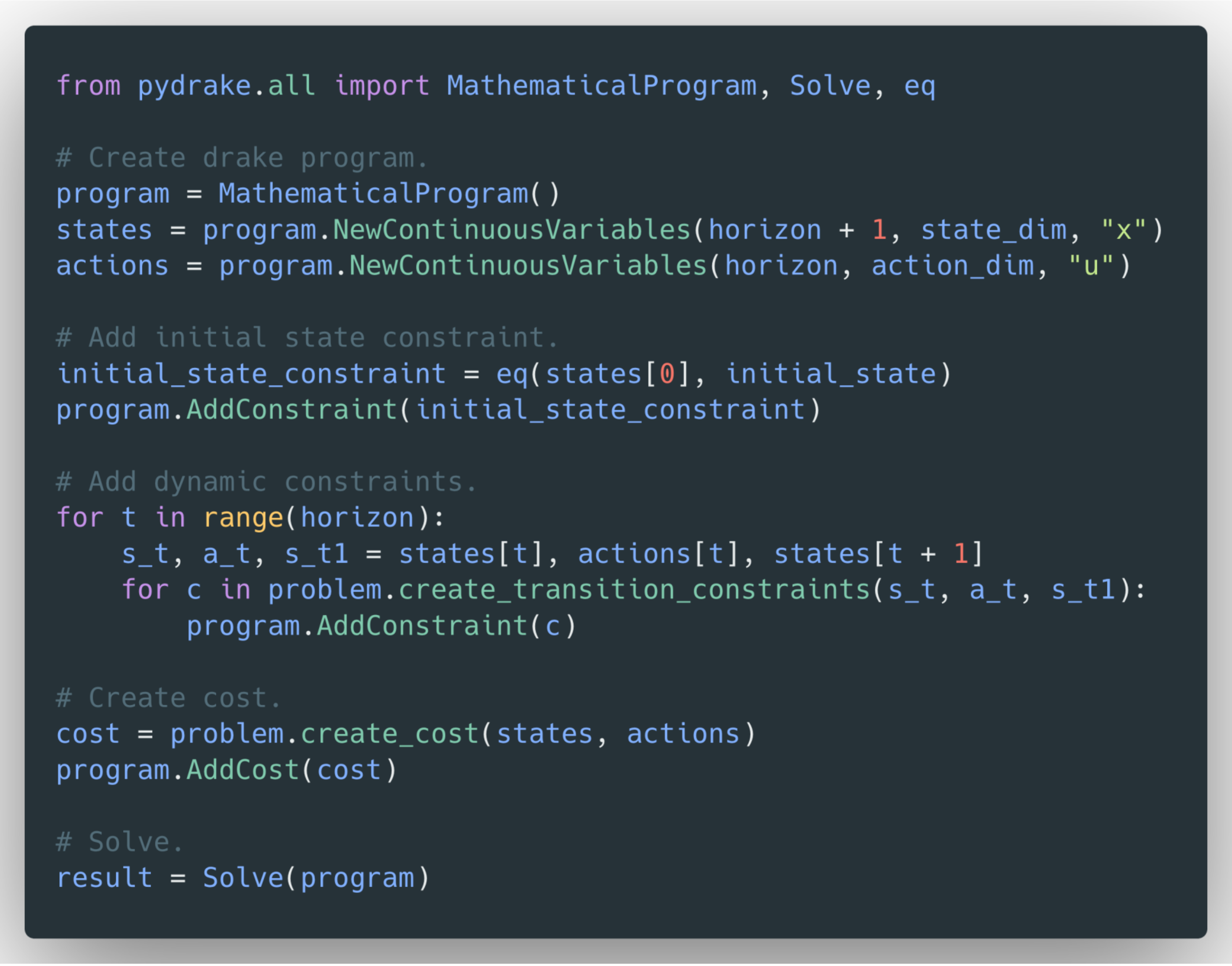
Objective

Equality constraints

Inequality constraints

- There are many methods for solving problems of this form
  - As usual, restricting the function classes leads to better methods

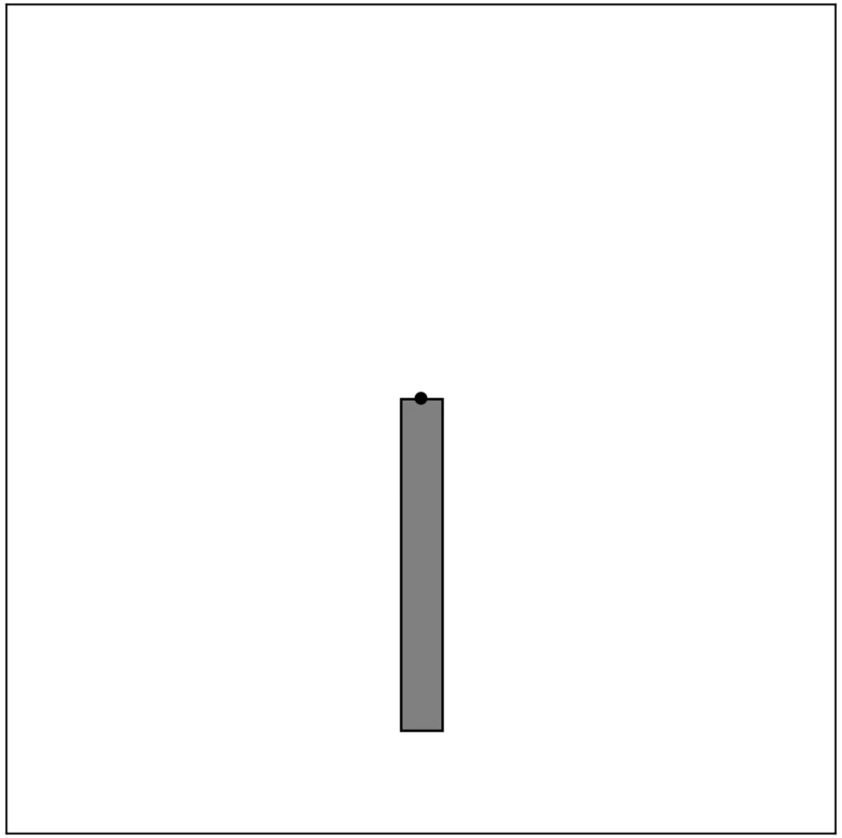
- For now, we will assume access to a black-box solver
  - In practice, we recommend SNOPT (via Drake)



```python
from pydrake.all import MathematicalProgram, Solve, eq

# Create drake program.
program = MathematicalProgram()
states = program.NewContinuousVariables(horizon + 1, state_dim, "x")
actions = program.NewContinuousVariables(horizon, action_dim, "u")

# Add initial state constraint.
initial_state_constraint = eq(states[0], initial_state)
program.AddConstraint(initial_state_constraint)

# Add dynamic constraints.
for t in range(horizon):
    s_t, a_t, s_t1 = states[t], actions[t], states[t + 1]
    for c in problem.create_transition_constraints(s_t, a_t, s_t1):
        program.AddConstraint(c)

# Create cost.
cost = problem.create_cost(states, actions)
program.AddCost(cost)

# Solve.
result = Solve(program)
```

SNOPT on Pendulum

With final state constraint and torque costs



Pretty impressive!

Peek Behind the Scenes

- SNOPT is a highly optimized version of **sequential quadratic programming**, a general method for constrained optimization

- SQP repeatedly creates a quadratic approximation of the objective around a nominal solution and then takes a Newton-Raphson step (much like we saw in DDP!)

SNOPT also leverages sparsity of equations for efficiency, like DDP

- To handle constraints, SQP uses the **Lagrange**:

$$\mathcal{L}(x, \lambda, \sigma) = f(x) + \lambda h(x) + \sigma g(x)$$

λ and σ are *Lagrange multipliers*

TrajOpt with SNOPT

What is optimized?	
Actions *a.k.a. indirect or shooting*	[✓ States and Actions *a.k.a. direct transcription or collocation*]

What derivatives (of dynamics and costs) are required?		
Zero-order	First-order	[✓ Second-order]

To what extent is the approach specific to *trajectory* optimization?		
Not really at all	[✓ Some]	A lot

Highest Level Takeaways

- Cast your trajectory optimization as general optimization
- But then leverage the trajectory structure to make computation more efficient (forward pass, backward pass, etc.)
- Use off-the-shelf optimization tools when possible

Good References

- Russ Tedrake's evolving notes: https://underactuated.mit.edu/trajopt.html
- Sergey Levine's lecture slides: https://rll.berkeley.edu/deeprlcoursesp17/docs/week_2_lecture_2_optimal_control.pdf
- The MuJoCo MPC (Predictive Sampling) paper: https://arxiv.org/pdf/2212.00541
- The constrained DDP paper: https://homes.cs.washington.edu/~todorov/papers/TassaICRA14.pdf

Let’s Play a Review Game

Bar trivia rules

- Break up into teams of 3-5
- Give your team a great name
- I will ask questions
- You will discuss quietly with your team
- Write down your answer
- Hold it up when I say so

Question 1

What are the three kinds of MDP time horizons?

Question 2

Suppose an MDP has N states, M actions, and for each state and action, there are at most K next states with nonzero probability.

What is the time complexity of one iteration of value iteration, assuming a good implementation?

Question 3

Consider the following “code”:

```
def solve_infinite_horizon_mdp(mdp):
    finite_horizon_mdp = convert_to_finite_horizon(mdp)
    policy = solve_finite_horizon_mdp(finite_horizon_mdp)
    # Guaranteed optimal policy for original mdp
    return policy
```

Is there some implementation of `convert_to_finite_horizon` that would make this code correct?

Question 4

Consider expectimax search in an MDP with 2 actions and 3 possible next states for each (state, action) pair.

Suppose we run expectimax search to a horizon of H=5.

Assuming that there are no redundant states, so trees == AODAGs, how many Bellman backups would we perform in total?

Question 5

Which of the following bandit exploration strategies are guaranteed to try all arms infinitely often in the limit?

1. Uniform random
2. Exploit only
3. Epsilon-greedy (for nontrivial epsilon)
4. UCB

You may select multiple.

Question 6

Which of the following is true about MCTS, but not about RTDP?

1. Requires only simulator access to MDP
2. Focuses on “promising” parts of AODAG
3. Adds one new state node at each iteration
4. Backpropagates values after each iteration
5. Uses rollout heuristic to estimate leaf node values
6. Uses greedy policy to select nodes to expand

You may select multiple.

Question 7

What is the time complexity of one step of state estimation for POMDPs?

Question 8

Describe a POMDP with 2 states and 2 observations where the corresponding Belief MDP has an infinite number of reachable states given any initial observation. Or explain why this is impossible.

Question 9

True or false: in classical planning, given an *optimal* heuristic, the number of nodes *expanded* by A* is equal to the number of actions in the output plan.

Question 10

Describe how one might use some of the trajectory optimization techniques we saw today to solve the motion planning problems that we saw last class. Write down key bullet point ideas, including *how* we would do this and *why/when* it would work well or not.

We'll use the "best" answer to break any ties!