Stanford **Spring 2024**

AA 203: Optimal and Learning-based Control
Homework #2
Due May 6 by 11:59 pm

Learning goals for this problem set:

Problem 1: Study sequential convex programming in nonlinear trajectory optimization and gain experience with CVXPY.

Problem 2: Familiarize the DP algorithm and appreciate the computational savings of DP versus an exhaustive search algorithm.

Problem 3: Gain experience with implementing LQR controllers by coding them "from scratch".

Problem 4: Apply iLQR for nonlinear trajectory optimization and stabilization, while using JAX for dynamics linearization.

Problem 5: Apply stochastic dynamic programming in stochastic environments by reasoning about expected utilities.

2.1 Cart-pole swing-up with limited actuation. Throughout this problem set, we will consider three instantiations of the classic "cart-pole" benchmark in which we attempt to balance an inverted pendulum upright on a cart. In this problem, we will tackle the challenging cart-pole "swing up" problem, in which the pendulum begins hanging downwards and is then brought to the upright position, with constraints on the control input that moves the cart horizontally. In practice, such control constraints often arise from motor limitations. The transcribed optimal control problem we would like to solve is

$$\begin{aligned} \underset{s,u}{\text{minimize}} \quad & \sum_{k=0}^{N-1} \Big((s_k - s_{\text{goal}})^\mathsf{T} Q (s_k - s_{\text{goal}}) + u_k^\mathsf{T} R u_k \Big) \\ \text{subject to} \quad & s_0 = \bar{s}_0 \\ & s_N = s_{\text{goal}} \\ & s_{k+1} = f_\mathsf{d}(s_k, u_k), \quad \forall k \in \{0, 1, \ldots, N-1\} \\ & u_k \in \mathcal{U}, \qquad\qquad \forall k \in \{0, 1, \ldots, N-1\} \end{aligned},$$

where $\bar{s}_0 \in \mathbb{R}^n$ is the initial state, $s_{\text{goal}} = (0, \pi, 0, 0)$ is the goal state (i.e., the upright position), $Q \succ 0$ and $R \succ 0$ are stage cost matrices, $\mathcal{U}$ is the control constraint set, and $f_\mathsf{d} : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ is a discretized form of the cart-pole dynamics.

However, this optimal control problem is non-convex due to the nonlinear equality constraints $s_{k+1} = f_\mathsf{d}(s_k, u_k)$; even if the dynamics were linear, this optimization cannot be solved with Riccati recursion due to the constraints on actuation. We will circumvent this issue with sequential convex programming (SCP). Recall that the key idea of SCP is to iteratively linearize the dynamics around an estimate $(s^{(i)}, u^{(i)})$ and solve a convex approximation of the optimal control problem near this trajectory, at each iteration i. Since linearization provides a good approximation to the nonlinear dynamics only in a small neighborhood around $(s^{(i)}, u^{(i)})$, the accuracy of the convex model may be poor if (s, u) deviates far from $(s^{(i)}, u^{(i)})$. To ensure smooth convergence, we consider a convex trust region around the state and input that is imposed as an additional constraint in the convex

optimization problem. Specifically, we consider a box around the nominal trajectory $(s^{(i)}, u^{(i)})$ described by

$$\|s_k - s_k^{(i)}\|_\infty \leq \rho, \ \forall k \in \{0, 1, \ldots, N\}$$
$$\|u_k - u_k^{(i)}\|_\infty \leq \rho, \ \forall k \in \{0, 1, \ldots, N-1\}$$

for some constant $\rho > 0$. Moreover, the terminal constraint $s_N = s_{\text{goal}}$ is difficult to enforce during each iteration of SCP; any sub-problem for which the trust region does not contain a swing-up maneuver will be infeasible. Thus, we will remove the terminal constraint $s_N = s_{\text{goal}}$ and add the terminal state cost term $(s_N - s_{\text{goal}})^\mathsf{T} P (s_N - s_{\text{goal}})$ as a proxy, where $P \succ 0$ has large entries.

In `cartpole_swingup_constrained.py`, you will use JAX and CVXPY to construct and solve a convex sub-problem at each SCP iteration. Carefully read *all* of the provided code.

(a) Derive the convex approximation of the optimal control problem around an iterate $(s^{(i)}, u^{(i)})$, including the control constraint set $\mathcal{U} \coloneqq [-u_{\max}, u_{\max}]$ with $u_{\max} > 0$, trust region constraints, and the terminal state cost term in place of the terminal state constraint. The convex problem should have linear equality constraints of the form $s_{k+1} = A_k s_k + B_k u_k + c_k$ obtained by linearizing the dynamics around $(s^{(i)}, u^{(i)})$. Derive A_k, B_k, and c_k in terms of $s^{(i)}$, $u^{(i)}$, and $f_\mathtt{d}$.

(b) Use JAX to complete the function `affinize` such that it computes A_k, B_k, and c_k from $s^{(i)}$, $u^{(i)}$, and $f_\mathtt{d}$ in two lines of code.

(c) Complete the function `scp_iteration`. Specifically, given the current iterate $(s^{(i)}, u^{(i)})$, use CVXPY to specify and solve the convex optimization problem you derived to obtain an updated solution $(s^{(i+1)}, u^{(i+1)})$.

(d) Run `cartpole_swingup_constrained.py`. Submit the generated state, control, and SCP cost plots. You should notice that the pendulum just reaches the goal state before the simulation ends; it would be more satisfying to swing the pendulum upright and then keep it there. In words, suggest a control scheme to do this.

Submit all of the requested plots and your completed version of `cartpole_swingup_constrained.py`.

(a) Derive the convex approximation of the optimal control problem around an iterate $(s^{(i)}, u^{(i)})$, including the control constraint set $\mathcal{U} := [-u_{\max}, u_{\max}]$ with $u_{\max} > 0$, trust region constraints, and the terminal state cost term in place of the terminal state constraint. The convex problem should have linear equality constraints of the form $s_{k+1} = A_k s_k + B_k u_k + c_k$ obtained by linearizing the dynamics around $(s^{(i)}, u^{(i)})$. Derive A_k, B_k, and c_k in terms of $s^{(i)}$, $u^{(i)}$, and f_{d}.

$$s_{k+1} = f_d(s_k, u_k)$$

$s^{(i)}, u^{(i)}$ refers to the entire trajectory

$$s_{k+1} = s_k + \int_{s_k} f_d(s_k, u_k)$$

$$s_{k+1} = \nabla_s f_d^T(s_k^{(i)}, u_k^{(i)}) \cdot (s_k - s_k^{(i)}) + \nabla_u f_d^T(s_k^{(i)}, u_k^{(i)}) \cdot (u_k - u_k^{(i)}) + f_d(s_k^{(i)}, u_k^{(i)})$$

$$= \nabla_s f_d(s_k^{(i)}, u_k^{(i)}) \cdot s_k + \nabla_u f_d(s_k^{(i)}, u_k^{(i)}) \cdot u_k - \nabla_s f_d(s_k^{(i)}, u_k^{(i)}) \cdot s_k^{(i)} - \nabla_u f_d(s_k^{(i)}, u_k^{(i)}) \cdot u_k^{(i)} + f_d(s_k^{(i)}, u_k^{(i)})$$

$$A_k = \nabla_s f_d(s_k^{(i)}, u_k^{(i)})$$

$$B_k = \nabla_u f_d(s_k^{(i)}, u_k^{(i)})$$

$$C = -A_k s_k^{(i)} - B_k u_k^{(i)} + f_d(s_k^{(i)}, u_k^{(i)})$$

$$\text{minimize } (S_N - S_{goal})^T P (S_N - S_{goal}) + \sum_{k=0}^{N-1} (S_k - S_{goal})^T Q (S_k - S_{goal}) + u_k^T R u_k$$

$$\text{subject to } S_{k+1} = f_d(S_k^{(i)}, u_k^{(i)}) + \nabla_S f_d(S_k^{(i)}, u_k^{(i)})(S_k - S_k^{(i)}) + \nabla_u f_d(S_k^{(i)}, u_k^{(i)})(u_k - u_k^{(i)}) \quad \forall k \in [0, N-1]$$

$$S_0 = [0, 0, 0, 0]$$

$$|u_k| \leq u_{max} \quad \forall k \in [0, N-1]$$

$$\| S_k - S_k^{(i)} \|_\infty \leq \rho \quad \forall k \in [0, N]$$

$$\| u_k - u_k^{(i)} \|_\infty \leq \rho \quad \forall k \in [0, N-1]$$

(d) We can run the open loop swing-up policy until the cartpole has reached a region that are well approximated linearly from the desired upright position and run a closed loop control around this upright position to keep the pole there.

2.2 Shortest path through a grid. Consider the shortest path problem in Figure 1 where it is only possible to travel to the right and the numbers represent the travel times for each segment. The control input is the decision to go "up" or "down" at each node. Given that state transitions are deterministic in a low-dimensional state space, we will use dynamic programming to develop a closed-loop policy for each node in the grid.

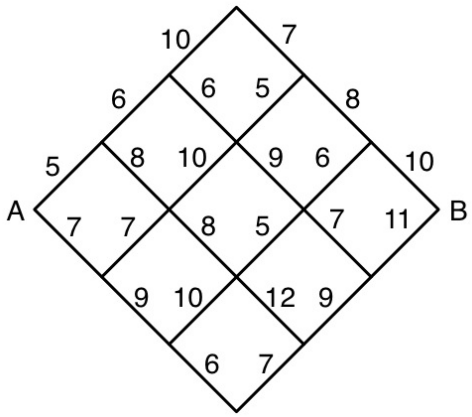
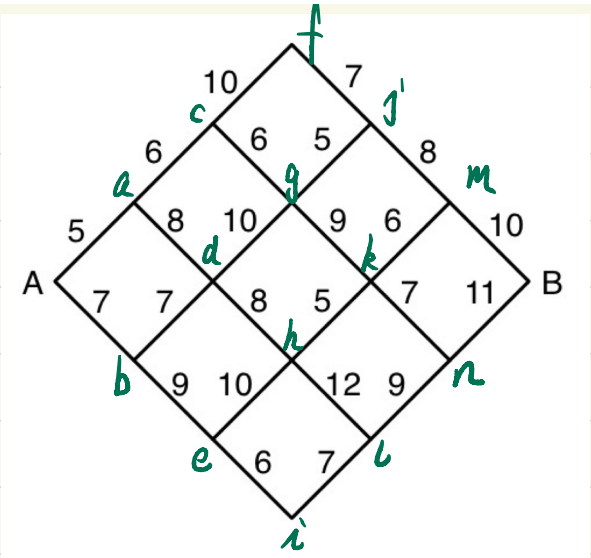


Figure 1: Shortest path problem on a grid.

(a) Use dynamic programming (DP) to find the shortest path from A to B.

(b) Consider a generalized version of the shortest path problem in Figure 1 where the grid has n segments on each side. Find the number of computations required by an exhaustive search algorithm (i.e., the number of routes that such an algorithm would need to evaluate) and the number of computations required by a DP algorithm (i.e., the number of DP evaluations). For example, for $n = 3$ as in Figure 1, an exhaustive search algorithm requires 20 computations, while the DP algorithm requires only 15.

(a) Use dynamic programming (DP) to find the shortest path from A to B.



$J(m) = 10$ $\quad J(n) = 11$

$J(j) = 8 + J(m) = 18$ $\quad J(k) = \min\{6 + J(m) = 16,\ 7 + J(n) = 18\} = 16$ $\quad J(l) = 9 + J(n) = 20$

$J(f) = 7 + J(j) = 25$ $\quad J(g) = \min\{5 + J(j) = 23,\ 9 + J(k) = 25\} = 23$ $\quad J(h) = \min\{5 + J(k) = 21,\ 12 + J(l) = 32\} = 21$

$J(i) = 7 + J(l) = 27$

$J(c) = \min\{10 + J(f) = 35,\ 6 + J(g) = 29\} = 29$ $\quad J(d) = \min\{10 + J(g) = 33,\ 8 + J(h) = 29\} = 29$

$J(e) = \min\{10 + J(h) = 31,\ 6 + J(i) = 33\} = 31$

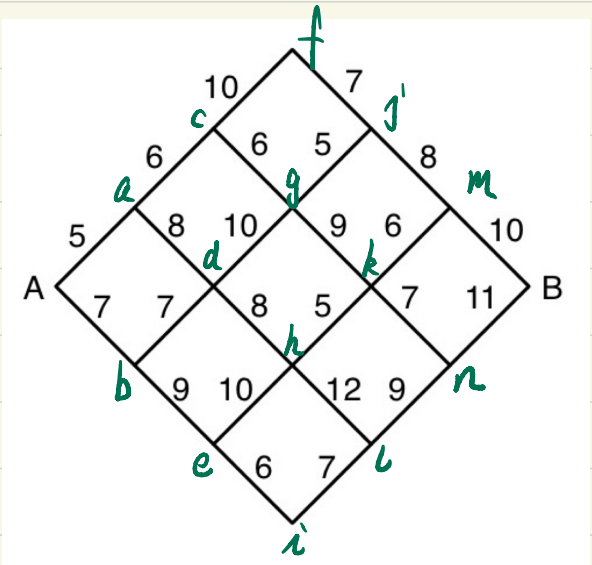
$$J(a) = \min\begin{cases} 6 + J(c) = 35 \\ 8 + J(d) = 37 \end{cases} = 35 \qquad J(b) = \min\begin{cases} 7 + J(d) = 36 \\ 9 + J(e) = 40 \end{cases} = 36$$

$$J(A) = \min\begin{cases} 5 + J(a) = 40 \\ 7 + J(b) = 43 \end{cases} = 40$$

Thus, the shortest path is

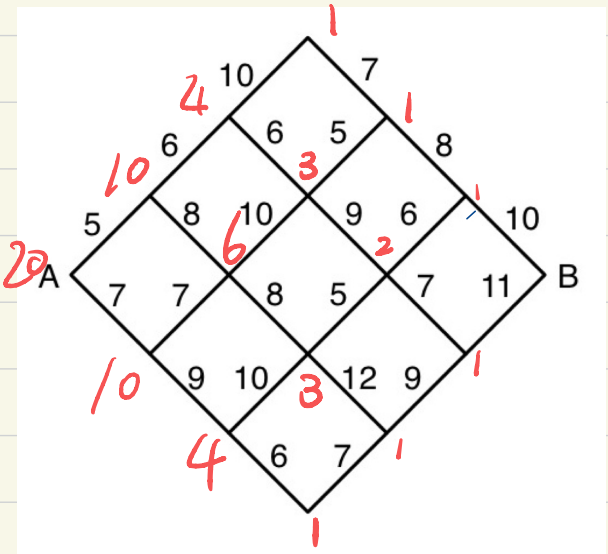
$A \to a \to c \to g \to j \to m \to B$

with a cost of 40



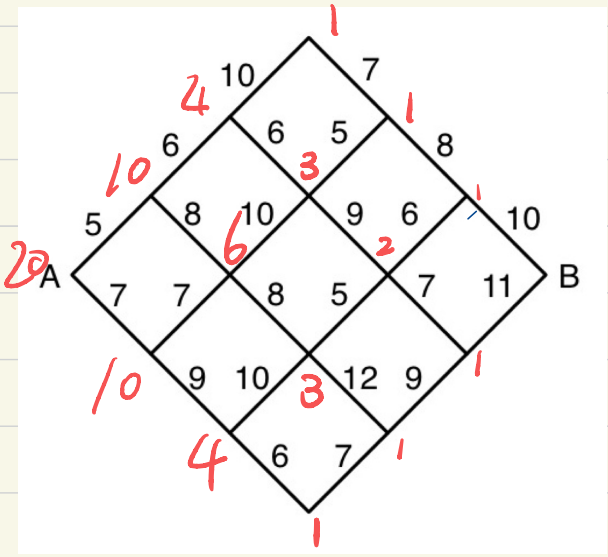
(b) Consider a generalized version of the shortest path problem in Figure 1 where the grid has n segments on each side. Find the number of computations required by an exhaustive search algorithm (i.e., the number of routes that such an algorithm would need to evaluate) and the number of computations required by a DP algorithm (i.e., the number of DP evaluations). For example, for $n = 3$ as in Figure 1, an exhaustive search algorithm requires 20 computations, while the DP algorithm requires only 15.

for $n=3$, the exhaustive search requires 20 evals
DP 15 evals



for $n=2$, the exhaustive search requires 6 evals
DP 8 evals

for $n=1$, the exhaustive search requires 2 evals
DP 3 evals



this is the Pascal triangle lying down

n
0	1
1	1 1
2	1 2 1
3	1 3 3 1
4	1 4 6 4 1
5	1 5 10 10 5 1
6	1 6 15 20 15 6 1

for pascal triangle
nth row and Kth col has value

$$\binom{n}{k}$$

for a grid with n segments on each side,
the number of routes in exhaustive search is then

$$\binom{2n}{n+1}$$

For DP, we need DP evaluations for each node prior to the final node.

For a $n \times n$ grid, there are in total $(n+1) \times (n+1)$ nodes excluding the destination node
we have $(n+1)^2 - 1$ DP evaluations to perform

2.3 Cart-pole balance. In this problem, we consider a new cart-pole formulation in which we will design a controller to balance the inverted pendulum by linearizing the dynamics around a single stationary point. Therefore, unlike the cart-pole swing-up problem, we will consider a trajectory initialized in a neighborhood about the final, upright state and the optimal control is a closed-form solution derived through Riccati recursion. This system has two degrees of freedom corresponding to the horizontal position x of the cart, and the angle θ of the pendulum (where $\theta = 0$ occurs when the pendulum is hanging straight downwards). We can apply a force $u \in \mathbb{R}$ to push the cart horizontally, where $u > 0$ corresponds to a force in the positive x-direction. With the state $s \coloneqq (x, \theta, \dot{x}, \dot{\theta}) \in \mathbb{R}^4$, we can write the continuous-time dynamics of the cart-pole system as

$$\dot{s} = f(s,u) = \begin{bmatrix} \dot{x} \\ \dot{\theta} \\ \frac{m_p(\ell\dot{\theta}^2 + g\cos\theta)\sin\theta + u}{m_c + m_p\sin^2\theta} \\ -\frac{(m_c+m_p)g\sin\theta + m_p\ell\dot{\theta}^2\sin\theta\cos\theta + u\cos\theta}{\ell(m_c+m_p\sin^2\theta)} \end{bmatrix},$$

where m_p is the mass of the pendulum, m_c is the mass of the cart, ℓ is the length of the pendulum, and g is the acceleration due to gravity. We can discretize the continuous-time dynamics using Euler integration with a fixed time step Δt to get the approximate discrete-time dynamics

$$s_{k+1} \approx s_k + \Delta t f(s_k, u_k),$$

where s_k and u_k are the state and control input, respectively, at time $t = k\Delta t$.

(a) Consider the upright state $\bar{s} \coloneqq (0, \pi, 0, 0)$ with $\bar{u} \coloneqq 0$, and define $\tilde{s}_k \coloneqq s_k - \bar{s}$. Linearizing the approximate discrete-time dynamics $s_{k+1} \approx s_k + \Delta t f(s_k, u_k)$ about $(\bar{s}, \bar{u})$ yields an approximate LTI system of the form

$$\tilde{s}_{k+1} \approx A\tilde{s}_k + Bu_k.$$

Express A and B in terms of m_p, m_c, ℓ, g, and Δt. You may use the fact that

$$\frac{\partial f}{\partial s}(\bar{s}, \bar{u}) = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{m_p g}{m_c} & 0 & 0 \\ 0 & \frac{(m_c+m_p)g}{m_c\ell} & 0 & 0 \end{bmatrix}, \quad \frac{\partial f}{\partial u}(\bar{s}, \bar{u}) = \begin{bmatrix} 0 \\ 0 \\ \frac{1}{m_c} \\ \frac{1}{m_c\ell} \end{bmatrix}.$$

We will design a stabilizing LQR controller for this discrete-time LTI system to solve

$$\begin{aligned} \underset{u}{\text{minimize}} \quad & \sum_{k=0}^{\infty}\left(\frac{1}{2}\tilde{s}_k^\mathsf{T} Q\tilde{s}_k + \frac{1}{2}u_k^\mathsf{T} R u_k\right) \\ \text{subject to} \quad & \tilde{s}_{k+1} = A\tilde{s}_k + Bu_k, \ \forall k \in \mathbb{N}_{\geq 0} \end{aligned},$$

for fixed $Q, R \succ 0$. Recall that after N iterations of the discrete-time Riccati recursion

$$\begin{aligned} K_k &= -(R + B^\mathsf{T} P_{k+1} B)^{-1} B^\mathsf{T} P_{k+1} A \\ P_k &= Q + A^\mathsf{T} P_{k+1}(A + BK_k) \end{aligned},$$

the cost-to-go matrices $\{P_k\}_{k=0}^N$ and the time-varying feedback gains $\{K_k\}_{k=0}^{N-1}$ describe the optimal LQR controller for a finite-horizon version of the problem above. If (A, B) is stabilizable, then these

What is the final P_k, P_N

@t=N, the cart pole should be up right with no motion
$P_N = 0$

iterates asymptotically converge to some $P_\infty \succ 0$ and K_∞. In fact, $J^*(s_0) \coloneqq (s_0 - \bar{s})^\mathsf{T} P_\infty (s_0 - \bar{s})$ is the *finite* optimal cost-to-go for any initialization s_0, and $u_k = K_\infty \tilde{s}_k$ is the optimal feedback policy, which happens to be linear and *time-invariant*[1].

(b) Write code to approximate P_∞ and K_∞ for the linearized, discretized cart-pole system by initializing $P_\infty = 0$ and then applying the Ricatti recursion until convergence with respect to the maximum element-wise norm condition $\|P_k - P_{k-1}\|_{\max} < 10^{-4}$. Use $m_p = 2$ kg, $m_c = 10$ kg, $\ell = 1$ m, $g = 9.81$ m/s^2, $\Delta t = 0.1$ s, $Q = I_4$, and $R = I_1$. Report the value of K_∞ with two decimal places for each entry.

(c) Write code to simulate the continuous-time, nonlinear cart-pole system with the linear feedback controller $u = K_\infty \tilde{s}$. Initialize the system at $s = (0, 3\pi/4, 0, 0)$, and use a controller sampling rate of 10 Hz. Plot each state variable and the control input over time on separate plots for $t \in [0, 30]$ (i.e., you should have five plots). For your own interest, we provide the function `animate_cartpole` in `animations.py` to create a video animation of the cart-pole over time.

Hint: Write a function `ds = cartpole(s,t,u)` that computes the state derivative `ds` for the continuous-time, nonlinear cart-pole dynamics. To simulate the cart-pole with the fixed control input `u[k]` from state `s[k]` at time `t[k]` to state `s[k+1]` at time `t[k+1]`, you can use the following Python code:

```
from scipy.integrate import odeint
s[k+1] = odeint(cartpole, s[k], t[k:k+2], (u[k],))[1]
```

Make sure to review the documentation for `odeint`.

(d) We will now use an LQR controller to track a time-varying trajectory. Specifically, we will aim to balance the pendulum upright (i.e., $\bar{\theta}(t) \equiv \pi$) while oscillating the position of the cart to track a desired reference $\bar{x}(t) = a \sin(2\pi t/T)$, where $a > 0$ and $T > 0$ are known constants.

i. Normally, as derived in class when applying LQR for trajectory tracking to nonlinear systems, you would have to re-linearize the system around the desired trajectory at each time step. Why is this not the case for this particular problem (i.e., why can you just reuse A and B)?

ii. Repeat part (c) for this case with $a = 10$ and $T = 10$, except this time initialize the system upright at $s(0) = (0, \pi, 0, 0)$. For each state plot, overlay the corresponding entry from the reference trajectory $\bar{s}(t)$.

iii. You may notice that this controller does not have good tracking performance. You could try increasing the state penalty matrix Q to, e.g., $Q = 10 I_4$. However, this should only improve tracking for $x(t)$ and $\dot{x}(t)$, while $\theta(t)$ and $\dot{\theta}(t)$ still oscillate around π and 0, respectively. What physical characteristic of the desired trajectory (or lack thereof) causes this to happen?

Submit all of the requested plots and your complete code.

[1] The infinite-horizon LQR problem also converges for fixed $Q \succeq 0$ and $R \succ 0$, as long as (A, B) is stabilizable and (A, Q) is detectable.

(a) Consider the upright state $\bar{s} := (0, \pi, 0, 0)$ with $\bar{u} := 0$, and define $\tilde{s}_k := s_k - \bar{s}$. Linearizing the approximate discrete-time dynamics $s_{k+1} \approx s_k + \Delta t f(s_k, u_k)$ about $(\bar{s}, \bar{u})$ yields an approximate LTI system of the form

$$\tilde{s}_{k+1} \approx A\tilde{s}_k + Bu_k.$$

Express A and B in terms of m_p, m_c, ℓ, g, and Δt. You may use the fact that

$$\frac{\partial f}{\partial s}(\bar{s}, \bar{u}) = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{m_p g}{m_c} & 0 & 0 \\ 0 & \frac{(m_c+m_p)g}{m_c \ell} & 0 & 0 \end{bmatrix}, \quad \frac{\partial f}{\partial u}(\bar{s}, \bar{u}) = \begin{bmatrix} 0 \\ 0 \\ \frac{1}{m_c} \\ \frac{1}{m_c \ell} \end{bmatrix}.$$

$$\dot{s} = f(s, u) = \begin{bmatrix} \dot{x} \\ \dot{\theta} \\ \frac{m_p(\ell\dot{\theta}^2 + g\cos\theta)\sin\theta + u}{m_c + m_p \sin^2\theta} \\ -\frac{(m_c+m_p)g\sin\theta + m_p\ell\dot{\theta}^2\sin\theta\cos\theta + u\cos\theta}{\ell(m_c + m_p\sin^2\theta)} \end{bmatrix}$$

sin π = 0, cos π = −1

where m_p is the mass of the pendulum, m_c is the mass of the cart, ℓ is the length of the pendulum, and g is the acceleration due to gravity. We can discretize the continuous-time dynamics using Euler integration with a fixed time step Δt to get the approximate discrete-time dynamics

$$s_{k+1} \approx s_k + \Delta t f(s_k, u_k),$$

where s_k and u_k are the state and control input, respectively, at time $t = k\Delta t$.

$\tilde{s}_{k+1} = s_{k+1} - \bar{s}$

$\approx s_k + \Delta t f(s_k, u_k) - \bar{s}$

Taylor expand the system dynamics around $\bar{s}$ and $\bar{u}$

$\dot{s} \approx f(\bar{s}, \bar{u}) + \nabla_s f(\bar{s}, \bar{u})(s - \bar{s}) + \nabla_u f(\bar{s}, \bar{u})(u - \bar{u})$

$f(\bar{s} = \begin{bmatrix} 0 \\ \pi \\ 0 \\ 0 \end{bmatrix}, u = 0) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

$\dot{s} = f(s, u) \approx \nabla_s f(\bar{s}, \bar{u})(s - \bar{s}) + \nabla_u f(\bar{s}, \bar{u})(u - \bar{u})$

$s_{k+1} \approx s_k + \Delta t f(s_k, u_k)$

$$\approx S_k + \Delta t \left[\nabla_s f(\bar{s}, \bar{u})(s_k - \bar{s}) + \nabla_u f(\bar{s}, \bar{u})(u - \bar{u}) \right]$$

minus $\bar{s}$ on both sides, we get

$$\tilde{s}_{k+1} = s_{k+1} - \bar{s} = (s_k - \bar{s}) + \Delta t \left[\nabla_s f(\bar{s}, \bar{u})(s_k - \bar{s}) + \nabla_u f(\bar{s}, \bar{u})(u - \bar{u}) \right]$$

swapping in definition of $\tilde{s}_k$, $\tilde{u}$

$$\tilde{s}_{k+1} = \tilde{s}_k + \Delta t \nabla f(\bar{s}, \bar{u}) \tilde{s}_k + \Delta t \nabla_u f(\bar{s}, \bar{u}) u$$

$$= (I + \Delta t \nabla f(\bar{s}, \bar{u})) \tilde{s}_k + \Delta t \nabla_u f(\bar{s}, \bar{u}) u$$

$$A = I + \Delta t \nabla_s f(\bar{s}, \bar{u})$$

$$= \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & \frac{m_p g}{m_c} \Delta t & 1 & 0 \\ 0 & \frac{(m_c + m_p) g}{m_c l} \Delta t & 0 & 1 \end{bmatrix}$$

$$B = \nabla_u f(\bar{s}, \bar{u}) = \begin{bmatrix} 0 \\ 0 \\ \frac{\Delta t}{m_c} \\ \frac{\Delta t}{m_c l} \end{bmatrix}$$

(c) Write code to simulate the continuous-time, nonlinear cart-pole system with the linear feedback controller $u = K_\infty \tilde{s}$. Initialize the system at $s = (0, 3\pi/4, 0, 0)$, and use a controller sampling rate of 10 Hz. Plot each state variable and the control input over time on separate plots for $t \in [0, 30]$ (i.e., you should have five plots). For your own interest, we provide the function `animate_cartpole` in `animations.py` to create a video animation of the cart-pole over time.

Hint: Write a function `ds = cartpole(s,t,u)` that computes the state derivative `ds` for the continuous-time, nonlinear cart-pole dynamics. To simulate the cart-pole with the fixed control input `u[k]` from state `s[k]` at time `t[k]` to state `s[k+1]` at time `t[k+1]`, you can use the following Python code:

```
from scipy.integrate import odeint
s[k+1] = odeint(cartpole, s[k], t[k:k+2], (u[k],))[1]
```

Make sure to review the documentation for `odeint`.

$$u = K_\infty \tilde{s} = K_\infty (s - \bar{s})$$

(d) We will now use an LQR controller to track a time-varying trajectory. Specifically, we will aim to balance the pendulum upright (i.e., $\bar{\theta}(t) \equiv \pi$) while oscillating the position of the cart to track a desired reference $\bar{x}(t) = a\sin(2\pi t/T)$, where $a > 0$ and $T > 0$ are known constants.

i. Normally, as derived in class when applying LQR for trajectory tracking to nonlinear systems, you would have to re-linearize the system around the desired trajectory at each time step. Why is this not the case for this particular problem (i.e., why can you just reuse A and B)?

ii. Repeat part (c) for this case with $a = 10$ and $T = 10$, except this time initialize the system upright at $s(0) = (0, \pi, 0, 0)$. For each state plot, overlay the corresponding entry from the reference trajectory $\bar{s}(t)$.

iii. You may notice that this controller does not have good tracking performance. You could try increasing the state penalty matrix Q to, e.g., $Q = 10I_4$. However, this should only improve tracking for $x(t)$ and $\dot{x}(t)$, while $\theta(t)$ and $\dot{\theta}(t)$ still oscillate around π and 0, respectively. What physical characteristic of the desired trajectory (or lack thereof) causes this to happen?

(ii) from tracking $\bar{s} = \begin{bmatrix} 0 \\ \pi \\ 0 \\ 0 \end{bmatrix}$

to tracking $\bar{s} = \begin{bmatrix} a\sin(2\pi t/T) \\ \pi \\ \frac{2\pi a}{T}\cos(2\pi t/T) \\ 0 \end{bmatrix}$

we are not changing how we are tracking $\theta = \pi$, $\dot{\theta} = 0$, $u = 0$

as the resulting jacobians w.r.t. s and u only have dependencies on θ, $\dot{\theta}$ and u, but not x and $\dot{x}$

$\nabla_s f$ and $\nabla_u f$ doesn't change w.r.t x and $\dot{x}$

then $A = I + \Delta t \nabla_s f$, $B = \nabla_u f$ } also stay the same w.r.t x and $\dot{x}$

$$\begin{bmatrix} \dot{x} \\ \dot{\theta} \\ \frac{m_p(\ell\dot{\theta}^2+g\cos\theta)\sin\theta+u}{m_c+m_p\sin^2\theta} \\ -\frac{(m_c+m_p)g\sin\theta+m_p\ell\dot{\theta}^2\sin\theta\cos\theta+u\cos\theta}{\ell(m_c+m_p\sin^2\theta)} \end{bmatrix}$$

$$\nabla_u f = \begin{bmatrix} 0 \\ 0 \\ \frac{1}{m_c+m_p\sin^2\theta} \\ -\frac{\cos\theta}{\ell(m_c+m_p\sin^2\theta)} \end{bmatrix} \Rightarrow B = \Delta t\, \nabla_u f$$

since we are still tracking $\theta = \pi$, $\dot{\theta} = 0$

there is no change to $\nabla_u f$

iii. See Code

iii. The desired trajectory has none zero minimal phase

i.e. the initial velocity for the trajectory is non-zero, whereas we have assumed a zero start,

2.4 Cart-pole swing-up. In this problem, we will implement a controller to solve the cart-pole "swing up" problem without limited actuation. Recall that in the swing-up problem, the pendulum begins hanging downwards and is then brought to the upright position. Unlike in the cart-pole balancing problem, it is no longer sufficient to linearize around a single stationary point. Therefore, we will formulate the optimal control problem into a convex sub-problem and iteratively solve for optimal perturbations from a reference trajectory. Unlike the case with limited actuation, the solution to each convex sub-problem can be solved with Riccati recursion, i.e., we will iteratively develop our solution using iterative LQR (iLQR) control. In this case iLQR control provides an elegant and easy to implement closed-loop policy. However, through the standard LQR formulation at each iteration, we generally cannot reason, at least directly, over constraints on the state or control space: in the case of limited actuation, we require the more flexible and sophisticated framework provided by SCP. We provide starter code in `cartpole_swingup.py`.

Recall that the cart-pole is a continuous-time system with dynamics of the form $\dot{s} = f(s, u)$. To compute the iLQR control law, we will consider the Euler discretized dynamics

$$s_{k+1} \approx f_{\mathsf{d}}(s_k, u_k) \coloneqq s_k + \Delta t f(s_k, u_k) \tag{1}$$

with time step $\Delta t > 0$. The provided code will then simulate this control law on the original continuous-time system.

(a) For a given operating point $(\bar{s}_k, \bar{u}_k)$, suppose we define the Jacobians

$$A_k \coloneqq \frac{\partial f_{\mathsf{d}}}{\partial s}(\bar{s}_k, \bar{u}_k), \quad B_k \coloneqq \frac{\partial f_{\mathsf{d}}}{\partial u}(\bar{s}_k, \bar{u}_k). \tag{2}$$

Use JAX in the function `linearize` to write a single line of code that computes A_k and B_k, given f_{d}, $\bar{s}_k$ and $\bar{u}_k$.

For our iLQR controller, we will use the quadratic cost function

$$J(s, u) \coloneqq \frac{1}{2}(s_N - s_{\text{goal}})^{\mathsf{T}} Q_N (s_N - s_{\text{goal}}) + \frac{1}{2} \sum_{k=0}^{N-1} \Big((s_k - s_{\text{goal}})^{\mathsf{T}} Q (s_k - s_{\text{goal}}) + u_k^{\mathsf{T}} R u_k \Big), \tag{3}$$

where s_{goal} is the goal state (i.e., the upright position). The entries of $Q_N \succ 0$ are chosen to be large so that the terminal cost acts as a soft terminal "constraint".

(b) Rewrite the cost function in terms of the deviations $\tilde{s}_N \coloneqq s_N - \bar{s}_N$, $\tilde{s}_k \coloneqq s_k - \bar{s}_k$, and $\tilde{u}_k \coloneqq u_k - \bar{u}_k$. This will result in a quadratic cost function with linear terms of the form $q_N^{\mathsf{T}} \tilde{s}_N$, $q_k^{\mathsf{T}} \tilde{s}_k$, and $r_k^{\mathsf{T}} \tilde{u}_k$; identify the vectors q_N, q_k, and r_k.

(c) Use NumPy to complete the iLQR controller code in the delineated section of the function `ilqr`. Specifically, your code must update the controller gain and offset terms $\{Y_k\}_{k=0}^{N-1}$ and $\{y_k\}_{k=0}^{N-1}$, respectively[2], the nominal trajectory $(\bar{s}, \bar{u})$, and the deviations $(\tilde{s}, \tilde{u})$.

(d) Towards the end of `cartpole_swingup.py`, fill in the delineated section of the code to either apply the open-loop iLQR control input or the closed-loop iLQR policy, depending on the value of the Boolean flag `closed_loop`. Run your code for both cases to simulate the system and generate plots of the state and control input over time. You should notice that the iLQR control sequence does not accomplish the task if applied open-loop. Submit both sets of plots (i.e., you should have 10 plots in total).

Submit all of the requested plots and your completed version of `cartpole_swingup.py`.

[2] In lecture, we labelled these terms as K_t and k_t, but here we try to avoid confusion with the discrete index k.

In LQR, we have

$$J_0^* = \frac{1}{2} x_N^T Q_N x_N + \frac{1}{2} \sum x_k^T Q x_k + u_k^T R u_k$$

for $x_{k+1} = A x_k + B u_k$

② we use recursion to find optimal $u_k = F_k x_k$ for each k

where $F_k = -(R + B_k^T P_{k+1} B_k)^{-1} B_k^T P_{k+1} A_k$

③ @ step k, then we have cost

$$J_k^* = \frac{1}{2} x_k^T P_k x_k$$

we calculate $P_k = Q + F_k^T R F_k + (A_k + B_k F_k)^T P_{k+1} (A_k + B_k F_k)$

④ then we can go back to step ② with P_k to calculate F_{k-1} & u_{k-1}, then to calculate P_{k-1}

⋮

For our iLQR controller, we will use the quadratic cost function

$$J(s,u) := \frac{1}{2}(s_N - s_{\text{goal}})^\mathsf{T} Q_N (s_N - s_{\text{goal}}) + \frac{1}{2}\sum_{k=0}^{N-1}\Big((s_k - s_{\text{goal}})^\mathsf{T} Q (s_k - s_{\text{goal}}) + u_k^\mathsf{T} R u_k\Big), \tag{3}$$

where s_{goal} is the goal state (i.e., the upright position). The entries of $Q_N \succ 0$ are chosen to be large so that the terminal cost acts as a soft terminal "constraint".

(b) Rewrite the cost function in terms of the deviations $\tilde{s}_N := s_N - \bar{s}_N$, $\tilde{s}_k := s_k - \bar{s}_k$, and $\tilde{u}_k := u_k - \bar{u}_k$. This will result in a quadratic cost function with linear terms of the form $q_N^\mathsf{T}\tilde{s}_N$, $q_k^\mathsf{T}\tilde{s}_k$, and $r_k^\mathsf{T}\tilde{u}_k$; identify the vectors q_N, q_k, and r_k.

(b)

$$\tilde{s}_N = s_N - \bar{s}_N \Rightarrow s_N = \bar{s}_N + \tilde{s}_N$$

$$\tilde{s}_k = s_k - \bar{s}_k \Rightarrow s_k = \bar{s}_k + \tilde{s}_k$$

$$\tilde{u}_k = u_k - \bar{u}_k \Rightarrow u_k = \bar{u}_k + \tilde{u}_k$$

$$J(s,u) = \frac{1}{2}(\bar{s}_N + \tilde{s}_N - s_{goal})^T Q_N (\bar{s}_N + \tilde{s}_N - s_{goal}) + \frac{1}{2}\sum_{k=0}^{N-1}\Big[(\bar{s}_k + \tilde{s}_k - s_{goal}) Q (\bar{s}_k + \tilde{s}_k - s_{goal}) + (\bar{u}_k + \tilde{u}_k)^T R (\bar{u}_k + \tilde{u}_k)\Big]$$

$$(\bar{s}_N + \tilde{s}_N - s_{goal})^T Q_N (\bar{s}_N + \tilde{s}_N - s_{goal})$$

$$= \bar{s}_N^T Q_N \bar{s}_N + \bar{s}_N^T Q_N \tilde{s}_N - \bar{s}_N^T Q_N s_{goal} \quad \text{Quadratic in } \tilde{s}_N$$

$$+ \tilde{s}_N^T Q_N \bar{s}_N + \tilde{s}_N^T Q_N \tilde{s}_N - \tilde{s}_N^T Q_N s_{goal} \quad \text{Linear in } \tilde{s}_N$$

$$- s_{goal}^T Q_N \bar{s}_N - s_{goal}^T Q_N \tilde{s}_N + s_{goal}^T Q_N s_{goal} \quad \text{Constant in } \tilde{s}_N$$

$$\bar{s}_N^T Q_N \tilde{s}_N + \tilde{s}_N^T Q_N \bar{s}_N - \tilde{s}_N^T Q_N s_{goal} - s_{goal}^T Q_N \tilde{s}_N$$

$$= 2 s_N^T Q_N \tilde{s}_N - 2 s_{goal}^T Q_N \tilde{s}_N$$

$$= 2(s_N^T Q_N - s_{goal}^T Q_N)\tilde{s}_N$$
$$= 2(s_N^T - s_{goal}^T) Q_N \tilde{s}_N$$

$$\Rightarrow \quad q_N^T := (s_N^T - s_{goal}^T) Q_N$$
$$q_N = Q_N^T (s_N - s_{goal})$$

$$(\bar{s}_k + \tilde{s}_k - s_{goal}) Q (\bar{s}_k + \tilde{s}_k - s_{goal})$$

$$= \bar{s}_k^T Q \bar{s}_k + \bar{s}_k^T Q \tilde{s}_k - \bar{s}_k^T Q s_{goal}$$
$$+ \tilde{s}_k^T Q \bar{s}_k + \tilde{s}_k^T Q \tilde{s}_k - \tilde{s}_k^T Q s_{goal}$$
$$- s_{goal}^T Q \bar{s}_k - s_{goal}^T Q \tilde{s}_k + s_{goal}^T Q s_{goal}$$

Quadratic in $\tilde{s}_k$

linear in $\tilde{s}_k$

constant in $\tilde{s}_k$

$$\bar{s}_k^T Q \tilde{s}_k + \tilde{s}_k^T Q \bar{s}_k - \tilde{s}_k^T Q s_{goal} - s_{goal}^T Q \tilde{s}_k$$
$$= 2\,\bar{s}_k^T Q \tilde{s}_k - 2\, s_{goal}^T Q \tilde{s}_k$$
$$= 2(s_k^T Q - s_{goal}^T Q)\tilde{s}_k = 2(s_k - s_{goal})^T Q \tilde{s}_k$$

$$\Rightarrow \quad q_k := Q^T (s_k - s_{goal})$$

$$(\bar{u}_k + \tilde{u}_k)^T R (\bar{u}_k + \tilde{u}_k)$$

$$= \bar{u}_k^T R \bar{u}_k + \bar{u}_k^T R \tilde{u}_k$$
$$+ \tilde{u}_k^T R \bar{u}_k + \tilde{u}_k^T R \tilde{u}_k$$
$$\bar{u}_k^T R \tilde{u}_k + \tilde{u}_k R \bar{u}_k$$
$$= 2\,\tilde{u}_k^T R \bar{u}_k$$
$$\Rightarrow \quad r_k = R^T u_k$$

Quadratic in $\tilde{u}_k$

linear in $\tilde{u}_k$

constant in $\tilde{u}_k$

$$J = \frac{1}{2}\tilde{s}_N^T Q_N \tilde{s}_N + q_N^T \tilde{s}_N + d_N$$
$$+ \sum \frac{1}{2}\tilde{s}_k^T Q_k \tilde{s}_k + q_k^T \tilde{s}_k + \frac{1}{2}\tilde{u}_k^T R \tilde{u}_k + r_k \tilde{u}_k + d_k$$

$$s_{k+1} \approx f_{\mathsf{d}}(s_k, u_k) := s_k + \Delta t f(s_k, u_k)$$

$$s_{k+1} = s_k + \Delta t\, f(s_k, u_k) + \Delta t$$

$$f_d(s_k, u_k) \approx f_d(\bar{s}_k, \bar{u}_k) + \underbrace{\nabla_s f_d(\bar{s}_k, \bar{u}_k)}_{A}(s - \bar{s}_k) + \underbrace{\nabla_u f_d(\bar{s}_k, \bar{u}_k)}_{B}(u_k - \bar{u}_k)$$
$$= f_d(\bar{s}, \bar{u}_k) + A(s_k - \bar{s}_k) + B(u_k - \bar{u}_k)$$

$$s_{k+1} \approx \underbrace{f_d(\bar{s}, \bar{u}_k)}_{\approx \bar{s}_{k+1}} + A(s_k - \bar{s}_k) + B(u_k - \bar{u}_k)$$

$$s_{k+1} - \bar{s}_{k+1} \approx A(s_k - \bar{s}_k) + B(u_k - \bar{u}_k)$$
$$\tilde{s}_{k+1} \approx A\tilde{s}_k + B\tilde{u}_k$$

$$J = \frac{1}{2}\left(\tilde{s}_N^T Q_N \tilde{s}_N + q_N^T \tilde{s}_N + d_N\right)$$
$$+ \frac{1}{2}\sum \tilde{s}_k^T Q\, \tilde{s}_k + q_k^T \tilde{s}_k + \tilde{u}_k^T R \tilde{u}_k + r_k \tilde{u}_k$$

$$J_N^* = \frac{1}{2}\left(\tilde{s}_N^T Q \tilde{s}_N + q_N^T \tilde{s}_N\right)$$

$$J_{N-1} = \frac{1}{2}\tilde{s}_{N-1}^T Q \tilde{s}_{N-1} + q_{N-1}^T \tilde{s}_{N-1} + \tilde{u}_{N-1}^T R \tilde{u}_{N-1} + r\tilde{u}_{N-1} + J_N^*$$

$$= \frac{1}{2}\left[\tilde{s}_{N-1}^T Q \tilde{s}_{N-1} + q_{N-1}^T \tilde{s}_{N-1} + \tilde{u}_{N-1}^T R \tilde{u}_{N-1} + r\tilde{u}_{N-1} + \tilde{s}_N^T Q s_N + q_N^T \tilde{s}_N\right]$$

$$= \frac{1}{2}\left[\tilde{s}_{N-1}^T Q \tilde{s}_{N-1} + q_{N-1}^T \tilde{s}_{N-1} + \tilde{u}_{N-1}^T R \tilde{u}_{N-1} + r\tilde{u}_{N-1}\right.$$
$$\left. + (A\tilde{s}_{N-1} + B\tilde{u}_{N-1})^T Q (A\tilde{s}_{N-1} + B\tilde{u}_{N-1}) + q_N^T (A\tilde{s}_{N-1} + B\tilde{u}_{N-1})\right]$$

$$\nabla_u J_{N-1} = R^T \tilde{u}_{N-1} + r_{N-1} + B^T Q^T (A\tilde{s}_{N-1} + B\tilde{u}_{N-1}) + B^T q_N \tilde{u}_{N-1}$$

Russ

$$Q_n + \begin{bmatrix} Q_{x,n} & Q_{u,n} \end{bmatrix} \begin{bmatrix} \delta x_n \\ \delta u_n \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \delta x_n & \delta u_n \end{bmatrix} \begin{bmatrix} Q_{xx,n} & Q_{ux,n}^T \\ Q_{ux,n} & Q_{uu,n} \end{bmatrix} \begin{bmatrix} \delta x_n \\ \delta u_n \end{bmatrix}$$

$$= Q_n + Q_{x,n}\delta x_n + Q_{u,n}\delta u_n + \frac{1}{2}\begin{bmatrix} \delta x_n^T & \delta u_n^T \end{bmatrix} \begin{bmatrix} Q_{xx,n}\delta x_n + Q_{ux,n}^T \delta u_n \\ Q_{ux,n}\delta x_n + Q_{uu,n}\delta u_n \end{bmatrix}$$

$$= Q_n + Q_{x,n}\delta x_n + Q_{u,n}\delta u_n + \frac{1}{2}\left(\delta x_n^T Q_{xx,n}\delta x_n + \delta x_n^T Q_{ux,n}^T \delta u_n + \delta u_n^T Q_{ux,n}\delta x_n + \delta u_n^T Q_{uu,n}\delta u_n \right)$$

$$Q_{x,n} = l_{x,n} + V_{x,n+1}^T f_{x,n}$$

$$Q_{u,n} = l_{u,n} + V_{x,n+1}^T f_{u,n}$$

$$Q_{xx,n} = l_{xx} + f_{x,n} V_{xx,n+1} f_{x,n}^T$$

$$Q_{ux,n}^T = l_{xu} + f_{x,n} V_{xx,n+1} f_{u,n}^T$$

$$Q_{ux,n} = l_{ux} + f_{u,n} V_{xx,n+1} f_{x,n}^T$$

$$Q_{uu,n} = l_{uu} + f_{u,n} V_{xx,n+1} f_{u,n}^T$$

$$V_{n+1} + V_{x,n+1}^T \delta x_{n+1} + \frac{1}{2} \delta x_{n+1}^T V_{xx} \delta x_{n+1}$$

$$= V_{n+1} + V_{x,n+1}^T \cdot \begin{bmatrix} f_{x,n} & f_{u,n} \end{bmatrix} \begin{bmatrix} \delta x_n \\ \delta u_n \end{bmatrix}$$

V is a scalar

$$+ \frac{1}{2} \begin{bmatrix} \delta x_n \\ \delta u_n \end{bmatrix}^T \begin{bmatrix} f_{x,n}^T \\ f_{u,n}^T \end{bmatrix} V_{xx,n+1} \begin{bmatrix} f_{x,n} & f_{u,n} \end{bmatrix} \begin{bmatrix} \delta x_n \\ \delta u_n \end{bmatrix}$$

$$= V_{n+1} + V_{x,n+1}^T \cdot (f_{x,n} \delta x_n + f_{u,n} \delta u_n)$$

$$+ \frac{1}{2} (\delta x_n^T f_{x,n}^T + \delta u_n^T f_{u,n}^T) V_{xx,n+1} (f_{x,n} \delta x_n + f_{u,n} \delta u_n)$$

$$= V_{n+1} + V_{x,n+1}^T f_{x,n} \delta x_n + V_{x,n+1}^T f_{u,n} \delta u_n$$

$$+ \frac{1}{2} \delta x_n^T f_{x,n}^T V_{xx,n+1} f_{x,n} \delta x_n$$

$$+ \frac{1}{2} \delta x_n^T f_{x,n}^T V_{xx,n+1} f_{u,n} \delta u_n$$

$$+ \frac{1}{2} \delta u_n^T f_{u,n}^T V_{xx,n+1} f_{x,n} \delta x_n$$

$$+ \frac{1}{2} \delta u_n^T f_{u,n}^T V_{xx,n+1} f_{u,n} \delta u_n$$

$$l_n + \begin{bmatrix} l_x \\ l_u \end{bmatrix}^T \begin{bmatrix} \delta x \\ \delta u \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \delta x \\ \delta u \end{bmatrix}^T \begin{bmatrix} l_{xx} & l_{xu} \\ l_{ux} & l_{uu} \end{bmatrix} \begin{bmatrix} \delta x \\ \delta u \end{bmatrix}$$

$$= l + l_x \delta x + l_u \delta u + \frac{1}{2} \begin{bmatrix} \delta x^T & \delta u^T \end{bmatrix} \begin{bmatrix} l_{xx}\delta x + l_{xu}\delta u \\ l_{ux}\delta x + l_{uu}\delta u \end{bmatrix}$$

$$= l + l_x \delta x + l_u \delta u + \frac{1}{2} \left(\delta x^T l_{xx} \delta x + \delta x^T l_{xu} \delta u + \delta u^T l_{ux} \delta x + \delta u^T l_{uu} \delta u \right)$$

$$Q \approx Q_n + Q_{x,n}^T \delta x_n + Q_{u,n}^T \delta u_n$$
$$+ \frac{1}{2}\left(\delta x_n^T Q_{xx,n} \delta x_n + \delta x_n^T Q_{ux,n}^T \delta u_n \right.$$
$$\left. + \delta u_n^T Q_{ux,n} \delta x_n + \delta u_n^T Q_{uu,n} \delta u_n \right)$$

$$\nabla_{\delta u} Q = 0 = Q_{u,n}^T + Q_{ux,n} \delta x_n + Q_{uu,n} \delta u_n$$
$$Q_{uu,n} \delta u_n = -Q_{u,n}^T - Q_{ux,n} \delta x_n$$
$$\delta u_n = \underbrace{-Q_{uu,n}^{-1} Q_{u,n}^T}_{k} \underbrace{- Q_{uu,n}^{-1} Q_{ux,n}}_{K} \delta x_n$$

Expected Cost Reduction

$$-Q_u^T \cdot k - \frac{1}{2} \cdot k^T Q_{uu} k$$

$$Q \approx Q_n + Q_{x,n}\delta x_n + Q_{u,n}\delta u_n + \frac{1}{2}\left(\delta x_n^T Q_{xx,n}\delta x_n + \delta x_n^T Q_{ux,n}^T \delta u_n + \delta u_n^T Q_{ux,n}\delta x_n + \delta u_n^T Q_{uu,n}\delta u_n\right)$$

plug in $\delta u^* = k + K\delta x$

$$Q_n + Q_{x,n}^T\delta x_n + Q_{u,n}^T(k + K\delta x) + \frac{1}{2}\delta x_n^T Q_{xx,n}\delta x_n + \delta x_n^T Q_{ux,n}^T(k+K\delta x) + \frac{1}{2}(k+K\delta x)^T Q_{uu,n}(k+K\delta x)$$

$$= Q_n + Q_{x,n}^T\delta x_n + Q_{u,n}^T k + Q_{u,n}^T K\delta x + \frac{1}{2}\delta x_n^T Q_{xx,n}\delta x_n + \delta x^T Q_{ux,n}^T k + \frac{1}{2}\delta x_n^T 2 Q_{ux,n}^T K\delta x + \frac{1}{2}k^T Q_{uu,n}k + \frac{1}{2}k^T Q_{uu,n}K\delta x + \frac{1}{2}\delta x^T K^T Q_{uu,n}k + \frac{1}{2}\delta x^T K^T Q_{uu,n}K\delta x$$

$\sim c$

$\sim \delta x$

$\sim \delta x^T(\cdot)\,\delta x$

$$V_x^T = (Q_{x,n}^T + Q_{u,n}^T K + Q_{ux,n}^T k + k^T Q_{uu,n}K)$$

$$V_x = Q_{x,n} + K^T Q_{u,n} + k^T Q_{ux,n} + K^T Q_{uu,n}k^T$$

$$V_{xx} = Q_{xx,n} + 2Q_{ux,n}^T K + K^T Q_{uu,n} K$$

$$Q + \begin{bmatrix} Q_x^T & Q_u^T \end{bmatrix} \begin{bmatrix} \delta x \\ \delta u \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \delta x \\ \delta u \end{bmatrix}^T \begin{bmatrix} Q_{xx} & Q_{ux}^T \\ Q_{ux} & Q_{uu} \end{bmatrix} \begin{bmatrix} \delta x \\ \delta u \end{bmatrix}$$

$$= Q + Q_x^T \delta x + Q_u^T \delta u + \frac{1}{2} \begin{bmatrix} \delta x \\ \delta u \end{bmatrix}^T \begin{bmatrix} Q_{xx}\delta x + Q_{ux}^T \delta u \\ Q_{ux}\delta x + Q_{uu}\delta u \end{bmatrix}$$

$$= Q + Q_x^T \delta x + Q_u^T \delta u + \frac{1}{2} \delta x^T Q_{xx} \delta x$$

$$\left.\begin{aligned} &+ \tfrac{1}{2} \delta x^T Q_{ux}^T \delta u \\ &+ \tfrac{1}{2} \delta u^T Q_{ux} \delta x \end{aligned}\right\} = \delta u^T Q_{ux} \delta x$$

$$+ \frac{1}{2} \delta u^T Q_{uu} \delta u$$

Let $\delta u = k + K \delta x$

$$Q + Q_x^T \delta x + Q_u^T (k + K\delta x) + \frac{1}{2} \delta x^T Q_{xx} \delta x$$

$$+ \delta x^T Q_{ux}^T (k + K \delta x)$$

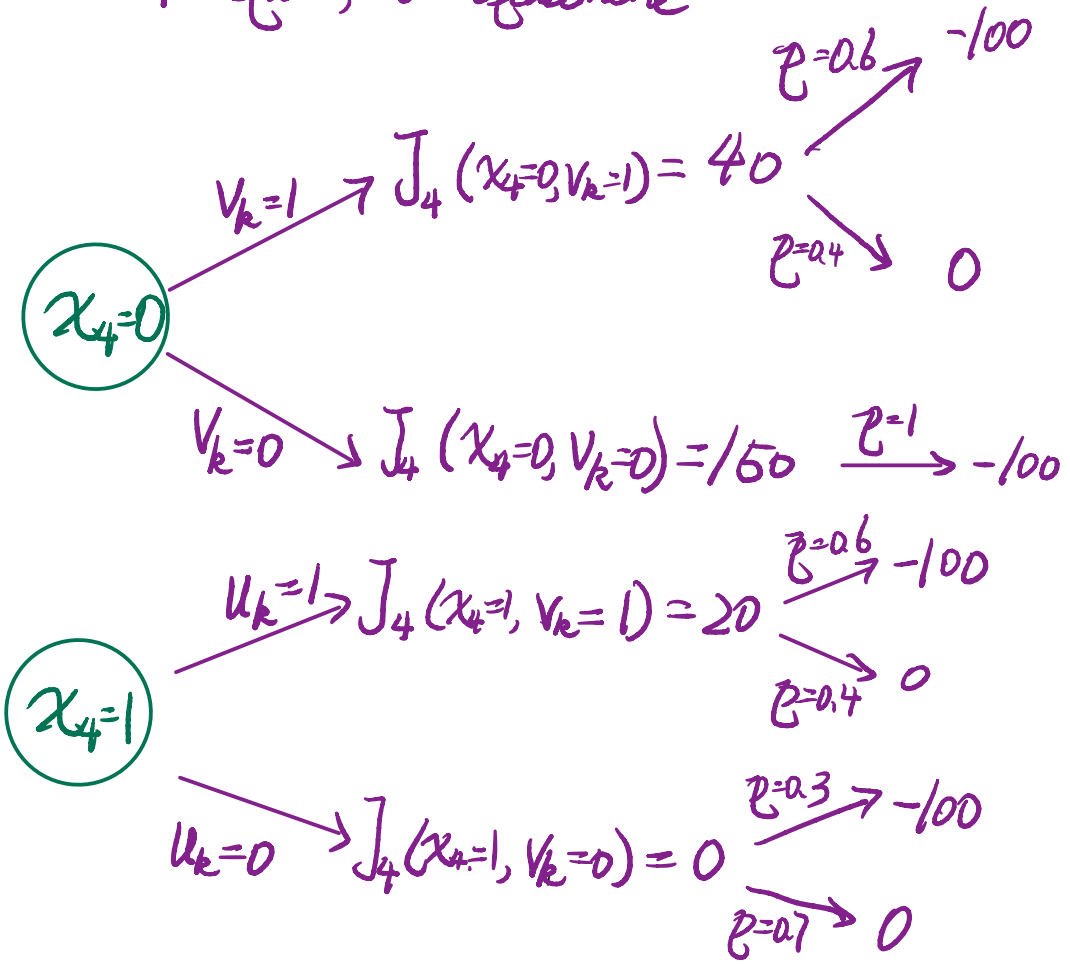
$$+ \frac{1}{2} \delta$$

2.5 Machine maintenance. Suppose we have a machine that is either running or is broken down. If it runs throughout one week, it makes a gross profit of \$100. If it fails during the week, gross profit is zero. If it is running at the start of the week and we perform preventive maintenance, the probability that it will fail during the week is 0.4. If we do not perform such maintenance, the probability of failure is 0.7. However, maintenance will cost \$20. When the machine is broken down at the start of the week, it may either be repaired at a cost of \$40, in which case it will fail during the week with a probability of 0.4, or it may be replaced at a cost of \$150 by a new machine that is guaranteed to run through its first week of operation. Find the optimal repair, replacement, and maintenance policy that maximizes total profit over four weeks, assuming a new machine at the start of the first week (that is guaranteed to run during the first week of operation).

x_k – state of machine at beginning of week k
└ 1 ready ; 0 broken

$x_k = 1$ u_k – preventive measure at the beginning of week k
└ 1 do it ; 0 do not do it

$x_k = 0 \Rightarrow v_k$ – repair/replacement at beginning of week k
└ 1 repair ; 0 replacement



For Week 4,

$$E[J_4(x_4=0, V_k=1)] = 40 + (-100)\,0.6 + 0 \cdot 0.4 = -20$$

$$E[J_4(x_4=0, V_k=0)] = 150 + (-100) \cdot 1 = 50$$

$$E[J_4(x_4=1, u_k=1)] = 20 + (-100)\,0.6 + 0 \cdot 0.4 = -40$$

$$E[J_4(x_4=1, u_k=0)] = 0 + (-100)\,0.3 + 0 \cdot 0.7 = -30$$

thus $J_4^*(x_4=1) = -40$ for $x_4 = 1$

$J_4^*(x_4=0) = -20$ for $x_4 = 0$

the implication for the machine state x_{k+1} at the start of next week is the same regardless of which one of the two policies we went down as a result of the machine state x_k for the prior week, i.e.

$$P(x_{k+1}=0) = 0.4 \qquad P(x_{k+1}=1) = 0.6$$

thus, at the end of Week 3, our expected value of optimal cost for Week 4 is

$$J_4^* = E_{x_4}[J^*(x_4)] = 0.4(-20) + 0.6(-40)$$
$$= -8 - 24$$
$$= -32$$

with policy $u_4 = 1$ for $x_4 = 0$
$u_4 = 1$ for $x_4 = 1$

by the same reasoning,

Week 2
Week 3
} has the stage wise cost as week 4

and week 1 has a fixed cost of +150 new machine purchase plus −100 assured income due to the guaranteed performance for one week

therefore, the optimal policy for minimizing cost over 4 weeks is the one that minimizes the cost over the last three weeks

we have shown that for all weeks except for the first week the optimal policy is $u_k = 1$ for $x_k = 0$

$u_k = 1$ for $x_k = 1$

with expected cost of −32

thus over the four week, the optimal expected cost is

$$J^* = 150 - 100 - 32 - 32 - 32$$
$$= -146$$

with the time-independent, but state dependent policy

$$\pi = \begin{cases} \text{repair } u=1 \text{ if broken } x=0 \\ \text{maintain } u=1 \text{ if functional } x=1 \end{cases}$$

HW2_Submission

May 6, 2024

0.1 Cart Pole Swingup Constrained

[ ]:
```
"""
Solution code for the problem "Cart-pole balance".

Autonomous Systems Lab (ASL), Stanford University
"""

import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

from animations import animate_cartpole

import pdb

# Constants
n = 4  # state dimension
m = 1  # control dimension
mp = 2.0  # pendulum mass
mc = 10.0  # cart mass
L = 1.0  # pendulum length
g = 9.81  # gravitational acceleration
dt = 0.1  # discretization time step
animate = False  # whether or not to animate results


def cartpole(s: np.ndarray, u: np.ndarray) -> np.ndarray:
    """Compute the cart-pole state derivative

    Args:
        s (np.ndarray): The cartpole state: [x, theta, x_dot, theta_dot], shape␣
 ↪(n,)
        u (np.ndarray): The cartpole control: [F_x], shape (m,)

    Returns:
        np.ndarray: The state derivative, shape (n,)
    """
```

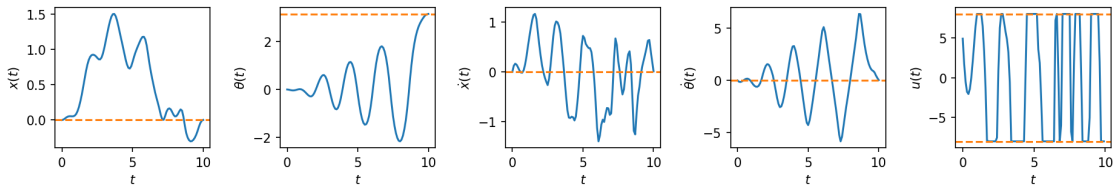
```
    x,  , dx, d  = s
    sin , cos  = np.sin( ), np.cos( )
    h = mc + mp * (sin **2)
    ds = np.array(
        [
            dx,
            d ,
            (mp * sin  * (L * (d **2) + g * cos ) + u[0]) / h,
            -((mc + mp) * g * sin  + mp * L * (d **2) * sin  * cos  + u[0] * cos )
            / (h * L),
        ]
    )
    return ds


def reference(t: float) -> np.ndarray:
    """Compute the reference state (s_bar) at time t

    Args:
        t (float): Evaluation time

    Returns:
        np.ndarray: Reference state, shape (n,)
    """
    a = 10.0  # Amplitude
    T = 10.0  # Period
    # breakpoint()
    # PART (d) ###################################################
    # INSTRUCTIONS: Compute the reference state for a given time
    # raise NotImplementedError()
    return np.array([a * np.sin(2*np.pi*t/T), np.pi, 2*np.pi*a/T * np.cos(2*np.
 ↪pi*t/T), 0]).T
    # END PART (d) ###############################################


def ricatti_recursion(
    A: np.ndarray, B: np.ndarray, Q: np.ndarray, R: np.ndarray
) -> np.ndarray:
    """Compute the gain matrix K through Ricatti recursion

    Args:
        A (np.ndarray): Dynamics matrix, shape (n, n)
        B (np.ndarray): Controls matrix, shape (n, m)
        Q (np.ndarray): State cost matrix, shape (n, n)
        R (np.ndarray): Control cost matrix, shape (m, m)

    Returns:
```

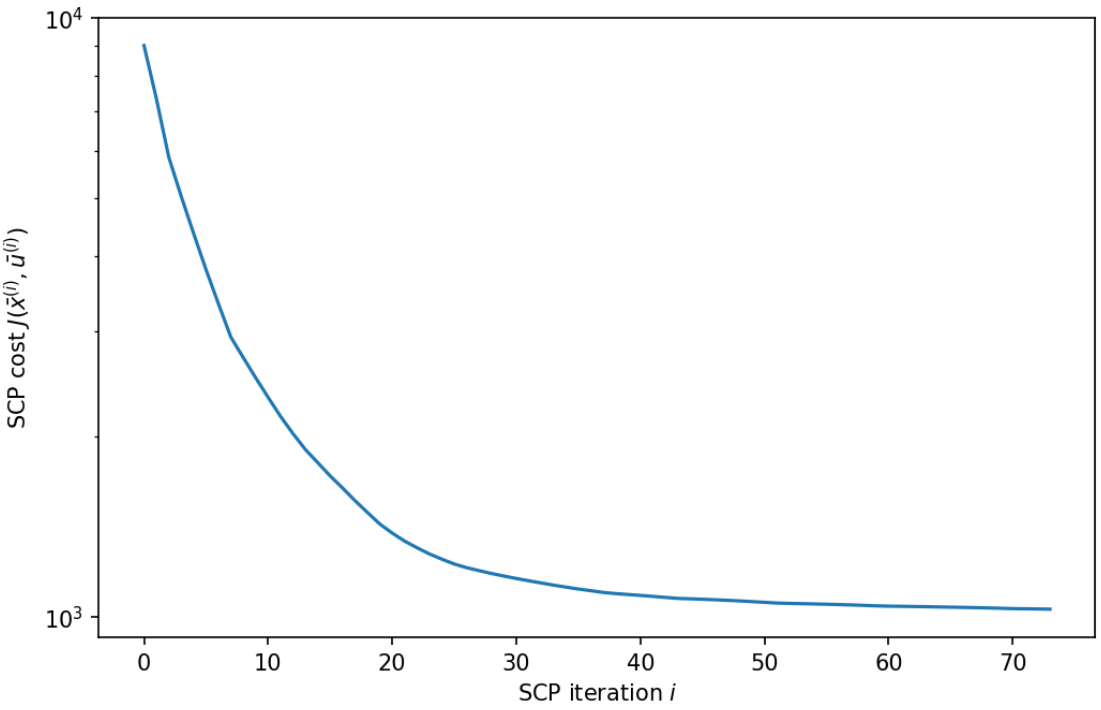
```
        np.ndarray: Gain matrix K, shape (m, n)
    """
    eps = 1e-4  # Riccati recursion convergence tolerance
    max_iters = 1000  # Riccati recursion maximum number of iterations
    P_prev = np.zeros((n, n))  # initialization
    converged = False
    for i in range(max_iters):
        # PART (b) ###################################################
        # INSTRUCTIONS: Apply the Ricatti equation until convergence
        K = -np.linalg.inv(R + B.T @ P_prev @ B) @ B.T @ P_prev @ A
        P_k = Q + A.T @ P_prev @ (A + B @ K)

        # termination condition
        if np.max(np.abs(P_prev-P_k)) < 1e-4:
            converged = True
            break
        else:
            P_prev = P_k
        # END PART (b) ###############################################
    if not converged:
        raise RuntimeError("Ricatti recursion did not converge!")
    print("K:", K)
    return K


def simulate(
    t: np.ndarray, s_ref: np.ndarray, u_ref: np.ndarray, s0: np.ndarray, K: np.
 ↪ndarray
) -> tuple[np.ndarray, np.ndarray]:
    """Simulate the cartpole

    Args:
        t (np.ndarray): Evaluation times, shape (num_timesteps,)
        s_ref (np.ndarray): Reference state s_bar, evaluated at each time t.␣
 ↪Shape (num_timesteps, n)
        u_ref (np.ndarray): Reference control u_bar, shape (m,)
        s0 (np.ndarray): Initial state, shape (n,)
        K (np.ndarray): Feedback gain matrix (Ricatti recursion result), shape␣
 ↪(m, n)

    Returns:
        tuple[np.ndarray, np.ndarray]: Tuple of:
            np.ndarray: The state history, shape (num_timesteps, n)
            np.ndarray: The control history, shape (num_timesteps, m)
    """

    def cartpole_wrapper(s, t, u):
```

```
        """Helper function to get cartpole() into a form preferred by odeint,␣
↪which expects t as the second arg"""
        return cartpole(s, u)

    # PART (c) ###################################################
    # INSTRUCTIONS: Complete the function to simulate the cartpole system
    # Hint: use the cartpole wrapper above with odeint

    # breakpoint()

    s = np.zeros((len(t), n))
    s[0,:] = s0

    u = np.zeros((len(t),m)) # [K@s0] #K @ s
    # print("State Variable: ", s_ref[0,:], s0)
    # breakpoint()
    u[0] = K @ (s0 - s_ref[0,:]).T - u_ref
    # breakpoint()
    for i, tk in enumerate(t[:-1]):

        # print(i, u[i,0], s[i,:])
        sol = odeint(cartpole_wrapper, s[i,:], t[0:0+2], (u[i],))
        # breakpoint()
        u[i+1] = K @ (sol[1] - s_ref[i]).T - u_ref
        s[i+1,:] = sol[1]

    # END PART (c) ###############################################
    return s, u


def compute_lti_matrices() -> tuple[np.ndarray, np.ndarray]:
    """Compute the linearized dynamics matrices A and B of the LTI system

    Returns:
        tuple[np.ndarray, np.ndarray]: Tuple of:
            np.ndarray: The A (dynamics) matrix, shape (n, n)
            np.ndarray: The B (controls) matrix, shape (n, m)
    """
    # PART (a) ###################################################
    df_ds = np.array(
        [
            [0, 0, 1, 0],
            [0, 0 ,0, 1],
            [0, mp*g/mc, 0, 0],
            [0, (mc+mp)*g/(mc*L), 0, 0]
        ]
    )
```

```
    df_du = np.array(
        [
            [0],
            [0],
            [1/mc],
            [1/(mc*L)]
        ]
    )

    # INSTRUCTIONS: Construct the A and B matrices
    A = np.eye(4) + dt * df_ds
    B = dt * df_du
    # END PART (a) #################################################
    return A, B


def plot_state_and_control_history(
    s: np.ndarray, u: np.ndarray, t: np.ndarray, s_ref: np.ndarray, name: str
) -> None:
    """Helper function for cartpole visualization

    Args:
        s (np.ndarray): State history, shape (num_timesteps, n)
        u (np.ndarray): Control history, shape (num_timesteps, m)
        t (np.ndarray): Times, shape (num_timesteps,)
        s_ref (np.ndarray): Reference state s_bar, evaluated at each time t.␣
 ↪Shape (num_timesteps, n)
        name (str): Filename prefix for saving figures
    """
    fig, axes = plt.subplots(1, n + m, dpi=150, figsize=(15, 2))
    plt.subplots_adjust(wspace=0.35)
    labels_s = (r"$x(t)$", r"$\theta(t)$", r"$\dot{x}(t)$",␣
 ↪r"$\dot{\theta}(t)$")
    labels_u = (r"$u(t)$",)
    for i in range(n):
        axes[i].plot(t, s[:, i])
        axes[i].plot(t, s_ref[:, i], "--")
        axes[i].set_xlabel(r"$t$")
        axes[i].set_ylabel(labels_s[i])
    for i in range(m):
        axes[n + i].plot(t, u[:, i])
        axes[n + i].set_xlabel(r"$t$")
        axes[n + i].set_ylabel(labels_u[i])
    plt.savefig(f"{name}.png", bbox_inches="tight")
    plt.show()
```

```python
    if animate:
        fig, ani = animate_cartpole(t, s[:, 0], s[:, 1])
        ani.save(f"{name}.mp4", writer="ffmpeg")
        plt.show()


def main():
    # Part A
    A, B = compute_lti_matrices()

    # Part B
    Q = np.eye(n) * 10  # state cost matrix
    R = np.eye(m)  # control cost matrix
    K = ricatti_recursion(A, B, Q, R)

    # Part C
    t = np.arange(0.0, 30.0, 1 / 10)
    s_ref = np.array([0.0, np.pi, 0.0, 0.0]) * np.ones((t.size, 1))
    u_ref = np.array([0.0])
    s0 = np.array([0.0, 3*np.pi/4, 0.0, 0.0])
    s, u = simulate(t, s_ref, u_ref, s0, K)
    plot_state_and_control_history(s, u, t, s_ref, "cartpole_balance")

    # Part D
    # Note: t, u_ref unchanged from part c
    s_ref = np.array([reference(ti) for ti in t])
    s0 = np.array([0.0, np.pi, 0.0, 0.0])
    s, u = simulate(t, s_ref, u_ref, s0, K)
    plot_state_and_control_history(s, u, t, s_ref, "cartpole_balance_tv")


if __name__ == "__main__":
    main()
```





0.2 Cart Pole Balancing

[1]:
```
"""
Solution code for the problem "Cart-pole balance".

Autonomous Systems Lab (ASL), Stanford University
"""

import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

from animations import animate_cartpole

import pdb

# Constants
n = 4  # state dimension
m = 1  # control dimension
mp = 2.0  # pendulum mass
mc = 10.0  # cart mass
L = 1.0  # pendulum length
g = 9.81  # gravitational acceleration
dt = 0.1  # discretization time step
animate = False  # whether or not to animate results
```

```python
def cartpole(s: np.ndarray, u: np.ndarray) -> np.ndarray:
    """Compute the cart-pole state derivative

    Args:
        s (np.ndarray): The cartpole state: [x, theta, x_dot, theta_dot], shape (n,)
        u (np.ndarray): The cartpole control: [F_x], shape (m,)

    Returns:
        np.ndarray: The state derivative, shape (n,)
    """
    x,  , dx, d  = s
    sin , cos  = np.sin( ), np.cos( )
    h = mc + mp * (sin **2)
    ds = np.array(
        [
            dx,
            d ,
            (mp * sin  * (L * (d **2) + g * cos ) + u[0]) / h,
            -((mc + mp) * g * sin  + mp * L * (d **2) * sin  * cos  + u[0] * cos )
            / (h * L),
        ]
    )
    return ds


def reference(t: float) -> np.ndarray:
    """Compute the reference state (s_bar) at time t

    Args:
        t (float): Evaluation time

    Returns:
        np.ndarray: Reference state, shape (n,)
    """
    a = 10.0  # Amplitude
    T = 10.0  # Period
    # breakpoint()
    # PART (d) ##################################################
    # INSTRUCTIONS: Compute the reference state for a given time
    # raise NotImplementedError()
    return np.array([a * np.sin(2*np.pi*t/T), np.pi, 2*np.pi*a/T * np.cos(2*np.pi*t/T), 0]).T
    # END PART (d) ##############################################
```

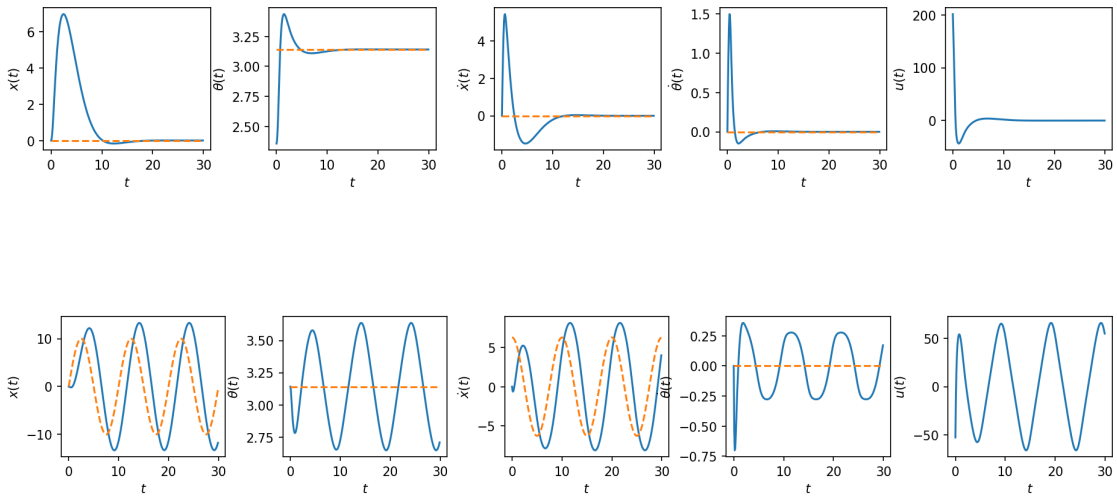
```python
def ricatti_recursion(
    A: np.ndarray, B: np.ndarray, Q: np.ndarray, R: np.ndarray
) -> np.ndarray:
    """Compute the gain matrix K through Ricatti recursion

    Args:
        A (np.ndarray): Dynamics matrix, shape (n, n)
        B (np.ndarray): Controls matrix, shape (n, m)
        Q (np.ndarray): State cost matrix, shape (n, n)
        R (np.ndarray): Control cost matrix, shape (m, m)

    Returns:
        np.ndarray: Gain matrix K, shape (m, n)
    """
    eps = 1e-4  # Riccati recursion convergence tolerance
    max_iters = 1000  # Riccati recursion maximum number of iterations
    P_prev = np.zeros((n, n))  # initialization
    converged = False
    for i in range(max_iters):
        # PART (b) ###########################################################
        # INSTRUCTIONS: Apply the Ricatti equation until convergence
        K = -np.linalg.inv(R + B.T @ P_prev @ B) @ B.T @ P_prev @ A
        P_k = Q + A.T @ P_prev @ (A + B @ K)

        # termination condition
        if np.max(np.abs(P_prev-P_k)) < 1e-4:
            converged = True
            break
        else:
            P_prev = P_k
        # END PART (b) #######################################################
    if not converged:
        raise RuntimeError("Ricatti recursion did not converge!")
    print("K:", K)
    return K


def simulate(
    t: np.ndarray, s_ref: np.ndarray, u_ref: np.ndarray, s0: np.ndarray, K: np.
 ↪ndarray
) -> tuple[np.ndarray, np.ndarray]:
    """Simulate the cartpole

    Args:
        t (np.ndarray): Evaluation times, shape (num_timesteps,)
        s_ref (np.ndarray): Reference state s_bar, evaluated at each time t.␣
 ↪Shape (num_timesteps, n)
```

```
        u_ref (np.ndarray): Reference control u_bar, shape (m,)
        s0 (np.ndarray): Initial state, shape (n,)
        K (np.ndarray): Feedback gain matrix (Ricatti recursion result), shape␣
↪(m, n)

    Returns:
        tuple[np.ndarray, np.ndarray]: Tuple of:
            np.ndarray: The state history, shape (num_timesteps, n)
            np.ndarray: The control history, shape (num_timesteps, m)
    """

    def cartpole_wrapper(s, t, u):
        """Helper function to get cartpole() into a form preferred by odeint,␣
↪which expects t as the second arg"""
        return cartpole(s, u)

    # PART (c) ###################################################
    # INSTRUCTIONS: Complete the function to simulate the cartpole system
    # Hint: use the cartpole wrapper above with odeint

    # breakpoint()

    s = np.zeros((len(t), n))
    s[0,:] = s0

    u = np.zeros((len(t),m)) # [K@s0] #K @ s
    # print("State Variable: ", s_ref[0,:], s0)
    # breakpoint()
    u[0] = K @ (s0 - s_ref[0,:]).T - u_ref
    # breakpoint()
    for i, tk in enumerate(t[:-1]):

        # print(i, u[i,0], s[i,:])
        sol = odeint(cartpole_wrapper, s[i,:], t[0:0+2], (u[i],))
        # breakpoint()
        u[i+1] = K @ (sol[1] - s_ref[i]).T - u_ref
        s[i+1,:] = sol[1]

    # END PART (c) ###############################################
    return s, u


def compute_lti_matrices() -> tuple[np.ndarray, np.ndarray]:
    """Compute the linearized dynamics matrices A and B of the LTI system

    Returns:
        tuple[np.ndarray, np.ndarray]: Tuple of:
```

```
            np.ndarray: The A (dynamics) matrix, shape (n, n)
            np.ndarray: The B (controls) matrix, shape (n, m)
    """
    # PART (a) ##################################################
    df_ds = np.array(
        [
            [0, 0, 1, 0],
            [0, 0 ,0, 1],
            [0, mp*g/mc, 0, 0],
            [0, (mc+mp)*g/(mc*L), 0, 0]
        ]
    )

    df_du = np.array(
        [
            [0],
            [0],
            [1/mc],
            [1/(mc*L)]
        ]
    )

    # INSTRUCTIONS: Construct the A and B matrices
    A = np.eye(4) + dt * df_ds
    B = dt * df_du
    # END PART (a) ##############################################
    return A, B


def plot_state_and_control_history(
    s: np.ndarray, u: np.ndarray, t: np.ndarray, s_ref: np.ndarray, name: str
) -> None:
    """Helper function for cartpole visualization

    Args:
        s (np.ndarray): State history, shape (num_timesteps, n)
        u (np.ndarray): Control history, shape (num_timesteps, m)
        t (np.ndarray): Times, shape (num_timesteps,)
        s_ref (np.ndarray): Reference state s_bar, evaluated at each time t.␣
 ↪Shape (num_timesteps, n)
        name (str): Filename prefix for saving figures
    """
    fig, axes = plt.subplots(1, n + m, dpi=150, figsize=(15, 2))
    plt.subplots_adjust(wspace=0.35)
    labels_s = (r"$x(t)$", r"$\theta(t)$", r"$\dot{x}(t)$",␣
 ↪r"$\dot{\theta}(t)$")
    labels_u = (r"$u(t)$",)
```

```
    for i in range(n):
        axes[i].plot(t, s[:, i])
        axes[i].plot(t, s_ref[:, i], "--")
        axes[i].set_xlabel(r"$t$")
        axes[i].set_ylabel(labels_s[i])
    for i in range(m):
        axes[n + i].plot(t, u[:, i])
        axes[n + i].set_xlabel(r"$t$")
        axes[n + i].set_ylabel(labels_u[i])
    plt.savefig(f"{name}.png", bbox_inches="tight")
    plt.show()

    if animate:
        fig, ani = animate_cartpole(t, s[:, 0], s[:, 1])
        ani.save(f"{name}.mp4", writer="ffmpeg")
        plt.show()


def main():
    # Part A
    A, B = compute_lti_matrices()

    # Part B
    Q = np.eye(n) * 10  # state cost matrix
    R = np.eye(m)  # control cost matrix
    K = ricatti_recursion(A, B, Q, R)

    # Part C
    t = np.arange(0.0, 30.0, 1 / 10)
    s_ref = np.array([0.0, np.pi, 0.0, 0.0]) * np.ones((t.size, 1))
    u_ref = np.array([0.0])
    s0 = np.array([0.0, 3*np.pi/4, 0.0, 0.0])
    s, u = simulate(t, s_ref, u_ref, s0, K)
    plot_state_and_control_history(s, u, t, s_ref, "cartpole_balance")

    # Part D
    # Note: t, u_ref unchanged from part c
    s_ref = np.array([reference(ti) for ti in t])
    s0 = np.array([0.0, np.pi, 0.0, 0.0])
    s, u = simulate(t, s_ref, u_ref, s0, K)
    plot_state_and_control_history(s, u, t, s_ref, "cartpole_balance_tv")


if __name__ == "__main__":
    main()
```

```
K: [[   2.26379564 -256.34294732    8.42679164  -76.04341843]]
```



[2]:
```
"""
Solution code for the problem "Cart-pole balance".

Autonomous Systems Lab (ASL), Stanford University
"""

import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

from animations import animate_cartpole

import pdb

# Constants
n = 4  # state dimension
m = 1  # control dimension
mp = 2.0  # pendulum mass
mc = 10.0  # cart mass
L = 1.0  # pendulum length
g = 9.81  # gravitational acceleration
dt = 0.1  # discretization time step
animate = False  # whether or not to animate results


def cartpole(s: np.ndarray, u: np.ndarray) -> np.ndarray:
    """Compute the cart-pole state derivative

    Args:
```

```
        s (np.ndarray): The cartpole state: [x, theta, x_dot, theta_dot], shape (n,)
        u (np.ndarray): The cartpole control: [F_x], shape (m,)

    Returns:
        np.ndarray: The state derivative, shape (n,)
    """
    x,  , dx, d  = s
    sin , cos  = np.sin( ), np.cos( )
    h = mc + mp * (sin **2)
    ds = np.array(
        [
            dx,
            d ,
            (mp * sin  * (L * (d **2) + g * cos ) + u[0]) / h,
            -((mc + mp) * g * sin  + mp * L * (d **2) * sin  * cos  + u[0] * cos )
            / (h * L),
        ]
    )
    return ds


def reference(t: float) -> np.ndarray:
    """Compute the reference state (s_bar) at time t

    Args:
        t (float): Evaluation time

    Returns:
        np.ndarray: Reference state, shape (n,)
    """
    a = 10.0  # Amplitude
    T = 10.0  # Period
    # breakpoint()
    # PART (d) ###################################################
    # INSTRUCTIONS: Compute the reference state for a given time
    # raise NotImplementedError()
    return np.array([a * np.sin(2*np.pi*t/T), np.pi, 2*np.pi*a/T * np.cos(2*np.pi*t/T), 0]).T
    # END PART (d) ###############################################


def ricatti_recursion(
    A: np.ndarray, B: np.ndarray, Q: np.ndarray, R: np.ndarray
) -> np.ndarray:
    """Compute the gain matrix K through Ricatti recursion
```

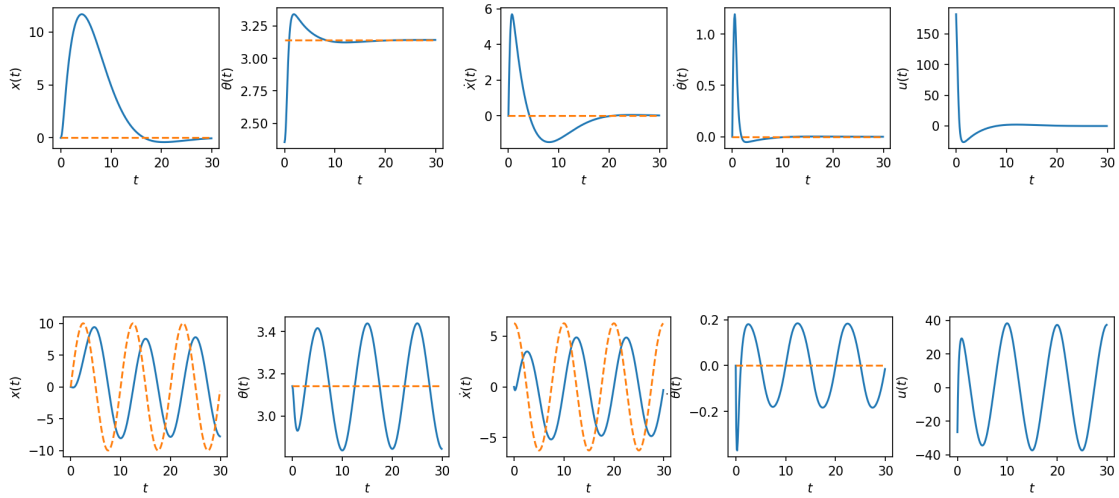
```
    Args:
        A (np.ndarray): Dynamics matrix, shape (n, n)
        B (np.ndarray): Controls matrix, shape (n, m)
        Q (np.ndarray): State cost matrix, shape (n, n)
        R (np.ndarray): Control cost matrix, shape (m, m)

    Returns:
        np.ndarray: Gain matrix K, shape (m, n)
    """
    eps = 1e-4  # Riccati recursion convergence tolerance
    max_iters = 1000  # Riccati recursion maximum number of iterations
    P_prev = np.zeros((n, n))  # initialization
    converged = False
    for i in range(max_iters):
        # PART (b) ###################################################
        # INSTRUCTIONS: Apply the Ricatti equation until convergence
        K = -np.linalg.inv(R + B.T @ P_prev @ B) @ B.T @ P_prev @ A
        P_k = Q + A.T @ P_prev @ (A + B @ K)

        # termination condition
        if np.max(np.abs(P_prev-P_k)) < 1e-4:
            converged = True
            break
        else:
            P_prev = P_k
        # END PART (b) ###############################################
    if not converged:
        raise RuntimeError("Ricatti recursion did not converge!")
    print("K:", K)
    return K


def simulate(
    t: np.ndarray, s_ref: np.ndarray, u_ref: np.ndarray, s0: np.ndarray, K: np.
 ↪ndarray
) -> tuple[np.ndarray, np.ndarray]:
    """Simulate the cartpole

    Args:
        t (np.ndarray): Evaluation times, shape (num_timesteps,)
        s_ref (np.ndarray): Reference state s_bar, evaluated at each time t.␣
 ↪Shape (num_timesteps, n)
        u_ref (np.ndarray): Reference control u_bar, shape (m,)
        s0 (np.ndarray): Initial state, shape (n,)
        K (np.ndarray): Feedback gain matrix (Ricatti recursion result), shape␣
 ↪(m, n)
```

```
    Returns:
        tuple[np.ndarray, np.ndarray]: Tuple of:
            np.ndarray: The state history, shape (num_timesteps, n)
            np.ndarray: The control history, shape (num_timesteps, m)
    """

    def cartpole_wrapper(s, t, u):
        """Helper function to get cartpole() into a form preferred by odeint,␣
↪which expects t as the second arg"""
        return cartpole(s, u)

    # PART (c) ###################################################
    # INSTRUCTIONS: Complete the function to simulate the cartpole system
    # Hint: use the cartpole wrapper above with odeint

    # breakpoint()

    s = np.zeros((len(t), n))
    s[0,:] = s0

    u = np.zeros((len(t),m)) # [K@s0] #K @ s
    # print("State Variable: ", s_ref[0,:], s0)
    # breakpoint()
    u[0] = K @ (s0 - s_ref[0,:]).T - u_ref
    # breakpoint()
    for i, tk in enumerate(t[:-1]):

        # print(i, u[i,0], s[i,:])
        sol = odeint(cartpole_wrapper, s[i,:], t[0:0+2], (u[i],))
        # breakpoint()
        u[i+1] = K @ (sol[1] - s_ref[i]).T - u_ref
        s[i+1,:] = sol[1]

    # END PART (c) ###############################################
    return s, u


def compute_lti_matrices() -> tuple[np.ndarray, np.ndarray]:
    """Compute the linearized dynamics matrices A and B of the LTI system

    Returns:
        tuple[np.ndarray, np.ndarray]: Tuple of:
            np.ndarray: The A (dynamics) matrix, shape (n, n)
            np.ndarray: The B (controls) matrix, shape (n, m)
    """
    # PART (a) ###################################################
    df_ds = np.array(
```

```
        [
            [0, 0, 1, 0],
            [0, 0 ,0, 1],
            [0, mp*g/mc, 0, 0],
            [0, (mc+mp)*g/(mc*L), 0, 0]
        ]
    )

    df_du = np.array(
        [
            [0],
            [0],
            [1/mc],
            [1/(mc*L)]
        ]
    )

    # INSTRUCTIONS: Construct the A and B matrices
    A = np.eye(4) + dt * df_ds
    B = dt * df_du
    # END PART (a) #################################################
    return A, B


def plot_state_and_control_history(
    s: np.ndarray, u: np.ndarray, t: np.ndarray, s_ref: np.ndarray, name: str
) -> None:
    """Helper function for cartpole visualization

    Args:
        s (np.ndarray): State history, shape (num_timesteps, n)
        u (np.ndarray): Control history, shape (num_timesteps, m)
        t (np.ndarray): Times, shape (num_timesteps,)
        s_ref (np.ndarray): Reference state s_bar, evaluated at each time t.␣
 ↪Shape (num_timesteps, n)
        name (str): Filename prefix for saving figures
    """
    fig, axes = plt.subplots(1, n + m, dpi=150, figsize=(15, 2))
    plt.subplots_adjust(wspace=0.35)
    labels_s = (r"$x(t)$", r"$\theta(t)$", r"$\dot{x}(t)$",␣
 ↪r"$\dot{\theta}(t)$")
    labels_u = (r"$u(t)$",)
    for i in range(n):
        axes[i].plot(t, s[:, i])
        axes[i].plot(t, s_ref[:, i], "--")
        axes[i].set_xlabel(r"$t$")
        axes[i].set_ylabel(labels_s[i])
```

```
    for i in range(m):
        axes[n + i].plot(t, u[:, i])
        axes[n + i].set_xlabel(r"$t$")
        axes[n + i].set_ylabel(labels_u[i])
    plt.savefig(f"{name}.png", bbox_inches="tight")
    plt.show()

    if animate:
        fig, ani = animate_cartpole(t, s[:, 0], s[:, 1])
        ani.save(f"{name}.mp4", writer="ffmpeg")
        plt.show()


def main():
    # Part A
    A, B = compute_lti_matrices()

    # Part B
    Q = np.eye(n) * 1  # state cost matrix
    R = np.eye(m)  # control cost matrix
    K = ricatti_recursion(A, B, Q, R)

    # Part C
    t = np.arange(0.0, 30.0, 1 / 10)
    s_ref = np.array([0.0, np.pi, 0.0, 0.0]) * np.ones((t.size, 1))
    u_ref = np.array([0.0])
    s0 = np.array([0.0, 3*np.pi/4, 0.0, 0.0])
    s, u = simulate(t, s_ref, u_ref, s0, K)
    plot_state_and_control_history(s, u, t, s_ref, "cartpole_balance")

    # Part D
    # Note: t, u_ref unchanged from part c
    s_ref = np.array([reference(ti) for ti in t])
    s0 = np.array([0.0, np.pi, 0.0, 0.0])
    s, u = simulate(t, s_ref, u_ref, s0, K)
    plot_state_and_control_history(s, u, t, s_ref, "cartpole_balance_tv")


if __name__ == "__main__":
    main()
```

```
K: [[   0.7291397  -231.85419273    4.21967187  -68.24742822]]
```



1 Cart Pole Balance

[ ]:
```
"""
Starter code for the problem "Cart-pole swing-up".

Autonomous Systems Lab (ASL), Stanford University
"""

import time

from animations import animate_cartpole

import jax
import jax.numpy as jnp

import matplotlib.pyplot as plt

import numpy as np

from scipy.integrate import odeint


def linearize(f, s, u):
    """Linearize the function `f(s, u)` around `(s, u)`.

    Arguments
    ---------
    f : callable
        A nonlinear function with call signature `f(s, u)`.
```

```
    s : numpy.ndarray
        The state (1-D).
    u : numpy.ndarray
        The control input (1-D).

    Returns
    -------
    A : numpy.ndarray
        The Jacobian of `f` at `(s, u)`, with respect to `s`.
    B : numpy.ndarray
        The Jacobian of `f` at `(s, u)`, with respect to `u`.
    """
    # WRITE YOUR CODE BELOW ###################################################
    # INSTRUCTIONS: Use JAX to compute `A` and `B` in one line.
    # raise NotImplementedError()
    A, B = jax.jacrev(f, argnums=(0, 1))(s, u)
    ###########################################################################
    return A, B


def ilqr(f, s0, s_goal, N, Q, R, QN, eps=1e-3, max_iters=1000):
    """Compute the iLQR set-point tracking solution.

    Arguments
    ---------
    f : callable
        A function describing the discrete-time dynamics, such that
        `s[k+1] = f(s[k], u[k])`.
    s0 : numpy.ndarray
        The initial state (1-D).
    s_goal : numpy.ndarray
        The goal state (1-D).
    N : int
        The time horizon of the LQR cost function.
    Q : numpy.ndarray
        The state cost matrix (2-D).
    R : numpy.ndarray
        The control cost matrix (2-D).
    QN : numpy.ndarray
        The terminal state cost matrix (2-D).
    eps : float, optional
        Termination threshold for iLQR.
    max_iters : int, optional
        Maximum number of iLQR iterations.

    Returns
    -------
```

```
    s_bar : numpy.ndarray
        A 2-D array where `s_bar[k]` is the nominal state at time step `k`,
        for `k = 0, 1, ..., N-1`
    u_bar : numpy.ndarray
        A 2-D array where `u_bar[k]` is the nominal control at time step `k`,
        for `k = 0, 1, ..., N-1`
    Y : numpy.ndarray
        A 3-D array where `Y[k]` is the matrix gain term of the iLQR control
        law at time step `k`, for `k = 0, 1, ..., N-1`
    y : numpy.ndarray
        A 2-D array where `y[k]` is the offset term of the iLQR control law
        at time step `k`, for `k = 0, 1, ..., N-1`
    """
    if max_iters <= 1:
        raise ValueError("Argument `max_iters` must be at least 1.")
    n = Q.shape[0]  # state dimension
    m = R.shape[0]  # control dimension

    # Initialize gains `Y` and offsets `y` for the policy
    Y = np.zeros((N, m, n))
    y = np.zeros((N, m))

    def rollout(x0, u_trj):
        x_trj = np.zeros((u_trj.shape[0] + 1, x0.shape[0]))
        # TODO: Define the rollout here and return the state trajectory x_trj:␣
↪[N, number of states]
        x_trj[0] = x0
        for i, u in enumerate(u_trj):
            x_trj[i+1] = fd(x_trj[i], u)
        return x_trj

    # Initialize the nominal trajectory `(s_bar, u_bar`), and the
    # deviations `(ds, du)`
    u_bar = np.random.rand(N, m)
    # s_bar[0] = s0
    s_bar = rollout(s0, u_bar)
    for k in range(N):
        s_bar[k + 1] = f(s_bar[k], u_bar[k])
    ds = np.zeros((N + 1, n))
    du = np.zeros((N, m))

    # Define Helper Function
    def forward_pass(x_trj, u_trj, k_trj, K_trj):
            x_trj_new = np.zeros(x_trj.shape)
            x_trj_new[0, :] = x_trj[0, :]
            u_trj_new = np.zeros(u_trj.shape)
            # TODO: Implement the forward pass here
```

```
            for n in range(u_trj.shape[0]):
                # Note, converting from deviation variable to actual value␣
↪variable
                u_trj_new[n,:] = u_trj[n,:] + k_trj[n,:] + K_trj[n,:] @␣
↪(x_trj_new[n,:]-x_trj[n,:])# Apply feedback law
                x_trj_new[n+1,:] = fd(x_trj_new[n,:], u_trj_new[n,:]) # Apply␣
↪dynamics
            return x_trj_new, u_trj_new

    @jax.jit
    def cost_stage(x, u):
        return 1/2 * ((x - s_goal).T @ Q @ (x-s_goal) + u.T @ R @ u)

    @jax.jit
    def stage(x, u):
        l = cost_stage
        l_x = jax.jacrev(l, argnums=0)
        l_u = jax.jacrev(l, argnums=1)
        l_xx = jax.jacrev(l_x, argnums=0)
        l_ux = jax.jacrev(l_u, argnums=0)
        l_uu = jax.jacrev(l_u, argnums=1)

        # f = fd
        f_x = jax.jacrev(fd, argnums=0)
        f_u = jax.jacrev(fd, argnums=1)


        return l_x(x, u), l_u(x, u), l_xx(x, u), l_ux(x, u), l_uu(x, u), f_x(x,␣
↪u), f_u(x, u)

    @jax.jit
    def cost_final(x):
        return 1/2 * (x - s_goal).T @ QN @ (x-s_goal)

    @jax.jit
    def final(x):
        l_final = cost_final
        l_final_x = jax.jacrev(l_final, argnums=0)
        l_final_xx = jax.jacrev(l_final_x, argnums=0)

        return  l_final_x(x), l_final_xx(x)

    @jax.jit
    def cost_trj(x_trj, u_trj):
        total = 0.0
        total = (
            cost_final(x_trj[-1])+
```

```
            jnp.sum(jnp.array([cost_stage(x, u) for x, u in zip(x_trj[:-1],␣
↪u_trj)]))
        )

        return total

    @jax.jit
    def gains(Q_uu, Q_u, Q_ux):
        Q_uu_inv = jnp.linalg.inv(Q_uu)
        # TODO: Implement the feedforward gain k and feedback gain K.
        k = - Q_uu_inv @ Q_u.T  #np.zeros(Q_u.shape)
        K = - Q_uu_inv @ Q_ux #np.zeros(Q_ux.shape)
        return k, K

    @jax.jit
    def V_terms(Q_x, Q_u, Q_xx, Q_ux, Q_uu, K, k):
        # TODO: Implement V_x and V_xx, hint: use the A.dot(B) function for␣
↪matrix multiplcation.
        V_x = Q_x + K.T @ Q_u + k.T @ Q_ux + K.T @ Q_uu @ k #np.zeros(Q_x.shape)
        # print(Q_xx.shape, Q_ux.T.shape, K.shape, K.T.shape, Q_ux.shape, K.
↪shape)  #np.zeros(Q_xx.shape)
        V_xx = Q_xx + 2 * Q_ux.T @ K + K.T @ Q_uu @ K  #np.zeros(Q_xx.shape)

        return V_x, V_xx

    @jax.jit
    def Q_terms(l_x, l_u, l_xx, l_ux, l_uu, f_x, f_u, V_x, V_xx):
        # TODO: Define the Q-terms here
        Q_x = l_x + V_x.T @ f_x #np.zeros(l_x.shape)
        Q_u = l_u + V_x.T @ f_u #np.zeros(l_u.shape)
        Q_xx = l_xx + f_x.T @ V_xx @ f_x #np.zeros(l_xx.shape)
        Q_ux = l_ux + f_u.T @ V_xx @ f_x #np.zeros(l_ux.shape)
        Q_uu = l_uu + f_u.T @ V_xx @ f_u #np.zeros(l_uu.shape)
        return Q_x, Q_u, Q_xx, Q_ux, Q_uu

    def backward_pass(x_trj, u_trj, regu=0):
        k_trj = np.zeros([u_trj.shape[0], u_trj.shape[1]])
        K_trj = np.zeros([u_trj.shape[0], u_trj.shape[1], x_trj.shape[1]])
        # expected_cost_redu = 0
        # TODO: Set terminal boundary condition here (V_x, V_xx)
        V_x, V_xx = final(x_trj[-1])

        for n in range(u_trj.shape[0] - 1, -1, -1):
            # TODO: First compute derivatives, then the Q-terms
            l_x, l_u, l_xx, l_ux, l_uu, f_x, f_u = stage(x_trj[n], u_trj[n])
```

```
            Q_x, Q_u, Q_xx, Q_ux, Q_uu = Q_terms(l_x, l_u, l_xx, l_ux, l_uu,␣
↪f_x, f_u, V_x, V_xx)

            # We add regularization to ensure that Q_uu is invertible and␣
↪nicely conditioned
            Q_uu_regu = Q_uu + np.eye(Q_uu.shape[0]) * regu
            k, K = gains(Q_uu_regu, Q_u, Q_ux)
            k_trj[n, :] = k
            K_trj[n, :, :] = K
            V_x, V_xx = V_terms(Q_x, Q_u, Q_xx, Q_ux, Q_uu, K, k)
            # expected_cost_redu += expected_cost_reduction(Q_u, Q_uu, k)
        return k_trj, K_trj#, expected_cost_redu


    # iLQR loop
    converged = False
    cost = np.inf
    for _ in range(max_iters):
        # Linearize the dynamics at each step `k` of `(s_bar, u_bar)`
        A, B = jax.vmap(linearize, in_axes=(None, 0, 0))(f, s_bar[:-1], u_bar)
        A, B = np.array(A), np.array(B)

        # PART (c) ###################################################################
        # INSTRUCTIONS: Update `Y`, `y`, `ds`, `du`, `s_bar`, and `u_bar`.
        # raise NotImplementedError()

        y, Y = backward_pass(s_bar, u_bar, regu=0)

        s_bar_new, u_bar_new = forward_pass(s_bar, u_bar, y, Y)

        print(cost_trj(s_bar_new, u_bar_new))

        du = u_bar_new - u_bar
        s_bar = s_bar_new
        u_bar = u_bar_new

        ##############################################################################

        if np.max(np.abs(du)) < eps:
            converged = True
            print('Converged')
            break
        # print('one more')
    if not converged:
        print("iLQR did not converge!")
    return s_bar, u_bar, Y, y
```

```
def cartpole(s, u):
    """Compute the cart-pole state derivative."""
    mp = 2.0  # pendulum mass
    mc = 10.0  # cart mass
    L = 1.0  # pendulum length
    g = 9.81  # gravitational acceleration

    x,  , dx, d  = s
    sin , cos  = jnp.sin( ), jnp.cos( )
    h = mc + mp * (sin **2)
    ds = jnp.array(
        [
            dx,
            d ,
            (mp * sin  * (L * (d **2) + g * cos ) + u[0]) / h,
            -((mc + mp) * g * sin  + mp * L * (d **2) * sin  * cos  + u[0] * cos )
            / (h * L),
        ]
    )
    return ds


# Define constants
n = 4  # state dimension
m = 1  # control dimension
Q = np.diag(np.array([10.0, 10.0, 2.0, 2.0]))  # state cost matrix
R = 1e-2 * np.eye(m)  # control cost matrix
QN = 1e2 * np.eye(n)  # terminal state cost matrix
s0 = np.array([0.0, 0.0, 0.0, 0.0])  # initial state
s_goal = np.array([0.0, np.pi, 0.0, 0.0])  # goal state
T = 10.0  # simulation time
dt = 0.1  # sampling time
animate = False  # flag for animation
closed_loop = False  # flag for closed-loop control

# Initialize continuous-time and discretized dynamics
f = jax.jit(cartpole)
fd = jax.jit(lambda s, u, dt=dt: s + dt * f(s, u))

# Compute the iLQR solution with the discretized dynamics
print("Computing iLQR solution ... ", end="", flush=True)
start = time.time()
t = np.arange(0.0, T, dt)
N = t.size - 1
s_bar, u_bar, Y, y = ilqr(fd, s0, s_goal, N, Q, R, QN)
print("done! ({:.2f} s)".format(time.time() - start), flush=True)
```

```
# Plot iLQR solution
# fig, axes = plt.subplots(1, n + m, dpi=150, figsize=(15, 2))
# plt.subplots_adjust(wspace=0.45)
# labels_s = (r"$x(t)$", r"$\theta(t)$", r"$\dot{x}(t)$", r"$\dot{\theta}(t)$")
# labels_u = (r"$u(t)$",)
# for i in range(n):
#     axes[i].plot(t, s_bar[:, i])
#     axes[i].set_xlabel(r"$t$")
#     axes[i].set_ylabel(labels_s[i])
# for i in range(m):
#     axes[n + i].plot(t[:-1], u_bar[:, i])
#     axes[n + i].set_xlabel(r"$t$")
#     axes[n + i].set_ylabel(labels_u[i])

# axes[1].axhline(np.pi, linestyle="--", color="tab:orange")

# Simulate on the true continuous-time system
print("Simulating ... ", end="", flush=True)
start = time.time()
s = np.zeros((N + 1, n))
u = np.zeros((N, m))
s[0] = s0
for k in range(N):
    # PART (d) ###################################################################
    # INSTRUCTIONS: Compute either the closed-loop or open-loop value of
    # `u[k]`, depending on the Boolean flag `closed_loop`.
    if closed_loop:
        u[k] = 0.0
        raise NotImplementedError()
    else:  # do open-loop control
        u[k] = u_bar[k]
        # raise NotImplementedError()
    ##############################################################################
    s[k + 1] = odeint(lambda s, t: f(s, u[k]), s[k], t[k : k + 2])[1]
print("done! ({:.2f} s)".format(time.time() - start), flush=True)

# Plot
fig, axes = plt.subplots(1, n + m, dpi=150, figsize=(15, 2))
plt.subplots_adjust(wspace=0.45)
labels_s = (r"$x(t)$", r"$\theta(t)$", r"$\dot{x}(t)$", r"$\dot{\theta}(t)$")
labels_u = (r"$u(t)$",)
for i in range(n):
    axes[i].plot(t, s[:, i])
    axes[i].set_xlabel(r"$t$")
    axes[i].set_ylabel(labels_s[i])
for i in range(m):
```

```
    axes[n + i].plot(t[:-1], u[:, i])
    axes[n + i].set_xlabel(r"$t$")
    axes[n + i].set_ylabel(labels_u[i])
if closed_loop:
    plt.savefig("cartpole_swingup_cl.png", bbox_inches="tight")
else:
    plt.savefig("cartpole_swingup_ol.png", bbox_inches="tight")
plt.show()

if animate:
    fig, ani = animate_cartpole(t, s[:, 0], s[:, 1])
    ani.save("cartpole_swingup.mp4", writer="ffmpeg")
    plt.show()
```

[ ]:

Iterative Linear Quadratic Regulator

In [1]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import pydrake.symbolic as sym
```

Iterative Linear Quadratic Regulator Derivation

In this exercise we will derive the iterative Linear Quadratic Regulator (iLQR) solving the following optimization problem.

$$\begin{aligned} \min_{\mathbf{u}[\cdot]} \quad & \ell_f(\mathbf{x}[N]) + \sum_{n=0}^{N-1} \ell(\mathbf{x}[n], \mathbf{u}[n]) \\ \text{subject to} \quad & \mathbf{x}[n+1] = (\mathbf{x}[n], \mathbf{u}[n]), \quad \forall n \in [0, N-1] \\ & \mathbf{x}[0] = \mathbf{x}_0 \end{aligned}$$

After completing this exercise you will be able to write your own MPC solver from scratch without any proprietary or third-party software (with the exception of auto-differentiation). You will derive all necessary equations yourself. While the iLQR algorithm will be capable of solving general model predictive control problems in the form described above, we will apply it to the control of a vehicle.

Vehicle Control Problem

Before we start the actual derivation of iLQR we will take a look at the vehicle dynamics and cost functions. The vehicle has the following continuous time dynamics and is controlled by longitudinal acceleration and steering velocity.

In [2]:
```
n_x = 5
n_u = 2


def car_continuous_dynamics(x, u):
    # x = [x position, y position, heading, speed, steering angle]
    # u = [acceleration, steering velocity]
    m = sym if x.dtype == object else np  # Check type for autodiff
    heading = x[2]
    v = x[3]
    steer = x[4]
    x_d = np.array(
        [v * m.cos(heading), v * m.sin(heading), v * m.tan(steer), u[0],
    )
    return x_d
```

Note that while the vehicle dynamics are in continuous time, our problem formulation is in discrete time. Define the general discrete time dynamics $\mathbf{f}$ with a simple Euler integrator in the next cell.

In [3]:
```
def discrete_dynamics(x, u):
    dt = 0.1
    # TODO: Fill in the Euler integrator below and return the next state

    x_next = x + dt * car_continuous_dynamics(x, u)
    return x_next
```

Given an initial state $\mathbf{x}_0$ and a guess of a control trajectory $\mathbf{u}[0:N-1]$ we roll out the state trajectory $x[0:N]$ until the time horizon N. Please complete the rollout function.

In [4]:
```
def rollout(x0, u_trj):
    x_trj = np.zeros((u_trj.shape[0] + 1, x0.shape[0]))
    # TODO: Define the rollout here and return the state trajectory x_trj
    x_trj[0] = x0
    for i, u in enumerate(u_trj):
        x_trj[i+1] = discrete_dynamics(x_trj[i], u)
    return x_trj


# Debug your implementation with this example code
N = 10
x0 = np.array([1, 0, 0, 1, 0])
u_trj = np.zeros((N - 1, n_u))
x_trj = rollout(x0, u_trj)
```

We define the stage cost function ℓ and final cost function ℓ_f. The goal of these cost functions is to drive the vehicle along a circle with radius r around the origin with a desired speed.s

$c_{circle} = \sqrt{x_0^2 + x_1^2} - r$ penalizes deviation from a circular route

$c_{speed} = (x_3 - v_{target})^2$ penalizes speed differences

$c_{control} = 0.1(u_0^2 + u_1^2)$ penalizes too much control up to a scaling factor

In [5]:
```
r = 2.0
v_target = 2.0
eps = 1e-6  # The derivative of sqrt(x) at x=0 is undefined. Avoid by sub


def cost_stage(x, u):
    m = sym if x.dtype == object else np  # Check type for autodiff

    c_circle = (m.sqrt(x[0] ** 2 + x[1] ** 2 + eps) - r) ** 2
    c_speed = (x[3] - v_target) ** 2
    c_control = (u[0] ** 2 + u[1] ** 2) * 0.1
    return c_circle + c_speed + c_control


def cost_final(x):
    m = sym if x.dtype == object else np  # Check type for autodiff
    c_circle = (m.sqrt(x[0] ** 2 + x[1] ** 2 + eps) - r) ** 2
    c_speed = (x[3] - v_target) ** 2
    return c_circle + c_speed
```

Your next task is to write the total cost function of the state and control trajectory. This is simply the sum of all stages over the control horizon and the objective from general problem formulation above.

In [6]:
```
def cost_trj(x_trj, u_trj):
    total = 0.0
    # TODO: Sum up all costs
    total = (
        cost_final(x_trj[-1])+
        np.sum([cost_stage(x, u) for x, u in zip(x_trj[:-1], u_trj)])
    )

    return total


# Debug your code
cost_trj(x_trj, u_trj)
```

Out[6]: 13.849995624574039

Bellman Recursion

Now that we are warmed up, let's derive the actual algorithm. We start with the Bellman equation known from lecture defining optimality in a recursively backwards in time.

$$V(\mathbf{x}[n]) = \min_{\mathbf{u}[n]} \quad \ell(\mathbf{x}[n], \mathbf{u}[n]) + V(\mathbf{x}[n+1])$$

You may have noticed that we neglected a couple of constraints of the original problem formulation. The fully equivalent formulation is

$$\begin{aligned} \min_{\mathbf{u}[n]} \quad & Q(\mathbf{x}[n], \mathbf{u}[n]), \quad \forall n \in [0, N-1] \\ \text{subject to} \quad & Q(\mathbf{x}[n], \mathbf{u}[n]) = \ell(\mathbf{x}[n], \mathbf{u}[n]) + V(\mathbf{x}[n+1]) \\ & V(\mathbf{x}[N]) = \ell_f(\mathbf{x}[N]) \\ & \mathbf{x}[n+1] = \mathbf{f}(\mathbf{x}[n], \mathbf{u}[n]), \\ & \mathbf{x}[0] = \mathbf{x}_0 \end{aligned}$$

The definition of a Q-function will become handy during the derivation of the algorithm.

The key idea of iLQR is simple: Approximate the dynamics linearly and the costs quadratically around a nominal trajectory. We will expand all terms of the Q-function accordingly and optimize the resulting quadratic equation for an optimal linear control law in closed form. We will see that by applying the Bellman equation recursively backwards in time, the value function remains a quadratic. The linear and quadratic approximations are computed around the nominal state $\bar{\mathbf{x}} = \mathbf{x} - \delta\mathbf{x}$ and the nominal control $\bar{\mathbf{u}} = \mathbf{u} - \delta\mathbf{u}$. After applying the Bellman equation backwards in time from time N to 0 (the backward pass), we will update the nominal controls $\bar{\mathbf{u}}$ and states $\bar{\mathbf{x}}$ by applying the computed linear feedback law from the backward pass and rolling out the dynamics from the initial state $\mathbf{x_0}$ to the final horizon N. Iterating between backwards

and forwards pass optimizes the control problem.

Q-function Expansion

Let's start by expanding all terms in the Q-function of the Bellman equation. The quadaratic cost function is

$$\ell(\mathbf{x}[n], \mathbf{u}[n]) \approx \ell_n + \begin{bmatrix} \ell_{\mathbf{x},n} \\ \ell_{\mathbf{u},n} \end{bmatrix}^T \begin{bmatrix} \delta\mathbf{x}[n] \\ \delta\mathbf{u}[n] \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \delta\mathbf{x}[n] \\ \delta\mathbf{u}[n] \end{bmatrix}^T \begin{bmatrix} \ell_{\mathbf{xx},n} & \ell_{\mathbf{ux},n}^T \\ \ell_{\mathbf{ux},n} & \ell_{\mathbf{uu},n} \end{bmatrix} \begin{bmatrix} \delta\mathbf{x}[n] \\ \delta\mathbf{u}[n] \end{bmatrix},$$

and the dynamics function is

$$x[n+1] = \mathbf{f}(\mathbf{x}[n], \mathbf{u}[n]) \approx \mathbf{f}_n + \begin{bmatrix} \mathbf{f}_{\mathbf{x},n} & \mathbf{f}_{\mathbf{u},n} \end{bmatrix} \begin{bmatrix} \delta\mathbf{x}[n] \\ \delta\mathbf{u}[n] \end{bmatrix}.$$

Here, $\ell = \ell(\bar{\mathbf{x}}, \bar{\mathbf{u}})$ and $\mathbf{f} = \mathbf{f}(\bar{\mathbf{x}}, \bar{\mathbf{u}})$. $\ell_{\mathbf{x}}, \ell_{\mathbf{u}}, \mathbf{f}_{\mathbf{x}}, \mathbf{f}_{\mathbf{u}}$ are the gradients and Jacobians evaluated at $\bar{\mathbf{x}}$ and $\bar{\mathbf{u}}$. $\ell_{\mathbf{xx}}, \ell_{\mathbf{ux}}, \ell_{\mathbf{uu}}$ are the Hessians at $\bar{\mathbf{x}}$ and $\bar{\mathbf{u}}$. The expansion of the final cost follows analogously. The code to evaluate all the derivative terms is:

In [7]:
```
class derivatives:
    def __init__(self, discrete_dynamics, cost_stage, cost_final, n_x, n_
        self.x_sym = np.array([sym.Variable("x_{}".format(i)) for i in ra
        self.u_sym = np.array([sym.Variable("u_{}".format(i)) for i in ra
        x = self.x_sym
        u = self.u_sym

        l = cost_stage(x, u)
        self.l_x = sym.Jacobian([l], x).ravel()
        self.l_u = sym.Jacobian([l], u).ravel()
        self.l_xx = sym.Jacobian(self.l_x, x)
        self.l_ux = sym.Jacobian(self.l_u, x)
        self.l_uu = sym.Jacobian(self.l_u, u)

        l_final = cost_final(x)
        self.l_final_x = sym.Jacobian([l_final], x).ravel()
        self.l_final_xx = sym.Jacobian(self.l_final_x, x)

        f = discrete_dynamics(x, u)
        self.f_x = sym.Jacobian(f, x)
        self.f_u = sym.Jacobian(f, u)

    def stage(self, x, u):
        env = {self.x_sym[i]: x[i] for i in range(x.shape[0])}
        env.update({self.u_sym[i]: u[i] for i in range(u.shape[0])})

        l_x = sym.Evaluate(self.l_x, env).ravel()
        l_u = sym.Evaluate(self.l_u, env).ravel()
        l_xx = sym.Evaluate(self.l_xx, env)
        l_ux = sym.Evaluate(self.l_ux, env)
        l_uu = sym.Evaluate(self.l_uu, env)

        f_x = sym.Evaluate(self.f_x, env)
        f_u = sym.Evaluate(self.f_u, env)

        return l_x, l_u, l_xx, l_ux, l_uu, f_x, f_u
```

```
    def final(self, x):
        env = {self.x_sym[i]: x[i] for i in range(x.shape[0])}

        l_final_x = sym.Evaluate(self.l_final_x, env).ravel()
        l_final_xx = sym.Evaluate(self.l_final_xx, env)

        return l_final_x, l_final_xx


derivs = derivatives(discrete_dynamics, cost_stage, cost_final, n_x, n_u)
# Test the output:
x = np.array([0, 0, 0, 0, 0])
u = np.array([0, 0])
# print(derivs.stage(x, u))
# print(derivs.final(x))
```

Expanding the second term of the Q-function of the Bellman equation, i.e. the value function at the next state $\mathbf{x}[n+1]$, to second order yields

$$V(\mathbf{x}[n+1]) \approx V_{n+1} + V_{\mathbf{x},n+1}^T \delta\mathbf{x}[n+1] + \frac{1}{2}\delta\mathbf{x}[n+1]^T V_{\mathbf{xx},n+1}\delta\mathbf{x}[n+1],$$

where $\delta\mathbf{x}[n+1]$ is given by

$$\begin{aligned}
\delta\mathbf{x}[n+1] &= \mathbf{x}[n+1] - \bar{\mathbf{x}}[n+1] \\
&= \mathbf{f}_n + \begin{bmatrix}\mathbf{f}_{\mathbf{x},n} & \mathbf{f}_{\mathbf{u},n}\end{bmatrix}\begin{bmatrix}\delta\mathbf{x}[n] \\ \delta\mathbf{u}[n]\end{bmatrix} - \bar{\mathbf{x}}[n+1] \\
&= \mathbf{f}_n + \begin{bmatrix}\mathbf{f}_{\mathbf{x},n} & \mathbf{f}_{\mathbf{u},n}\end{bmatrix}\begin{bmatrix}\delta\mathbf{x}[n] \\ \delta\mathbf{u}[n]\end{bmatrix} - \mathbf{f}(\bar{\mathbf{x}}[n], \bar{\mathbf{u}}[n]) \\
&= \begin{bmatrix}\mathbf{f}_{\mathbf{x},n} & \mathbf{f}_{\mathbf{u},n}\end{bmatrix}\begin{bmatrix}\delta\mathbf{x}[n] \\ \delta\mathbf{u}[n]\end{bmatrix}.
\end{aligned}$$

We have now expanded all terms of the Bellman equation and can regroup them in the form of

$$\begin{aligned}
Q(\mathbf{x}[n], \mathbf{u}[n]) &\approx \ell_n + \begin{bmatrix}\ell_{\mathbf{x},n} \\ \ell_{\mathbf{u},n}\end{bmatrix}^T \begin{bmatrix}\delta\mathbf{x}[n] \\ \delta\mathbf{u}[n]\end{bmatrix} + \frac{1}{2}\begin{bmatrix}\delta\mathbf{x}[n] \\ \delta\mathbf{u}[n]\end{bmatrix}^T \begin{bmatrix}\ell_{\mathbf{xx},n} & \ell_{\mathbf{ux},n}^T \\ \ell_{\mathbf{ux},n} & \ell_{\mathbf{uu},n}\end{bmatrix}\begin{bmatrix}\delta\mathbf{x}[n] \\ \delta\mathbf{u}[n]\end{bmatrix}, \\
&\quad + V_{n+1} + V_{\mathbf{x},n+1}^T \delta\mathbf{x}[n+1] + \frac{1}{2}\delta\mathbf{x}[n+1]^T V_{\mathbf{xx},n+1}\delta\mathbf{x}[n+1], \\
&= Q_n + \begin{bmatrix}Q_{\mathbf{x},n} \\ Q_{\mathbf{u},n}\end{bmatrix}^T \begin{bmatrix}\delta\mathbf{x}[n] \\ \delta\mathbf{u}[n]\end{bmatrix} + \frac{1}{2}\begin{bmatrix}\delta\mathbf{x}[n] \\ \delta\mathbf{u}[n]\end{bmatrix}^T \begin{bmatrix}Q_{\mathbf{xx},n} & Q_{\mathbf{ux},n}^T \\ Q_{\mathbf{ux},n} & Q_{\mathbf{uu},n}\end{bmatrix}\begin{bmatrix}\delta\mathbf{x}[n] \\ \delta\mathbf{u}[n]\end{bmatrix}
\end{aligned}$$

Find $Q_{\mathbf{x},n}, Q_{\mathbf{u},n}, Q_{\mathbf{xx},n}, Q_{\mathbf{ux},n}, Q_{\mathbf{uu},n}$ in terms of ℓ and $\mathbf{f}$ and their expansions by collecitng coefficients in $(\cdot)\delta\mathbf{x}[n]$, $(\cdot)\delta\mathbf{u}[n]$, $1/2\delta\mathbf{x}[n]^T(\cdot)\delta\mathbf{x}[n]$, and similar. Write your results in the corresponding function below.

In [8]:
```
def Q_terms(l_x, l_u, l_xx, l_ux, l_uu, f_x, f_u, V_x, V_xx):
    # TODO: Define the Q-terms here
    Q_x = l_x + V_x.T @ f_x #np.zeros(l_x.shape)
    Q_u = l_u + V_x.T @ f_u #np.zeros(l_u.shape)
    Q_xx = l_xx + f_x.T @ V_xx @ f_x #np.zeros(l_xx.shape)
    Q_ux = l_ux + f_u.T @ V_xx @ f_x #np.zeros(l_ux.shape)
    Q_uu = l_uu + f_u.T @ V_xx @ f_u #np.zeros(l_uu.shape)
```

```
    return Q_x, Q_u, Q_xx, Q_ux, Q_uu
```

Q-function Optimization and Optimal Linear Control Law

Amazing! Now that we have the Q-function in quadratic form, we can optimize for the optimal control gains in closed form. The original formulation, i.e. optimizing over $\mathbf{u}[n]$,

$$\min_{\mathbf{u}[n]} \quad Q(\mathbf{x}[n], \mathbf{u}[n]),$$

is equivalent to optimzing over $\delta\mathbf{u}[n]$.

$$\delta\mathbf{u}[n]^* = \text{argmin}_{\delta\mathbf{u}[n]} \quad Q_n + \begin{bmatrix} Q_{\mathbf{x},n} \\ Q_{\mathbf{u},n} \end{bmatrix}^T \begin{bmatrix} \delta\mathbf{x}[n] \\ \delta\mathbf{u}[n] \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \delta\mathbf{x}[n] \\ \delta\mathbf{u}[n] \end{bmatrix}^T \begin{bmatrix} Q_{\mathbf{xx},n} & Q_{\mathbf{ux},n}^T \\ Q_{\mathbf{ux},n} & Q_{\mathbf{uu},n} \end{bmatrix} \begin{bmatrix} \delta$$

It turns out that the optimal control is linear in $\delta\mathbf{x}[n]$. Solve the quadratic optimization analytically and derive equations for the feedforward gains k and feedback gains K. Implement the function below. Hint: You do not need to compute $Q_{\mathbf{uu}}^{-1}$ by hand.

In [9]:
```
def gains(Q_uu, Q_u, Q_ux):
    Q_uu_inv = np.linalg.inv(Q_uu)
    # TODO: Implement the feedforward gain k and feedback gain K.
    k = - Q_uu_inv @ Q_u.T  #np.zeros(Q_u.shape)
    K = - Q_uu_inv @ Q_ux #np.zeros(Q_ux.shape)
    return k, K
```

Value Function Backward Update

We are almost done! We need to derive the backwards update equation for the value function. We simply plugin the optimal control $\delta\mathbf{u}[n]^* = k + K\delta\mathbf{x}[n]$ back into the Q-function which yields the value function

$$V(\mathbf{x}[n]) \approx V_n + V_{\mathbf{x},n}^T \delta\mathbf{x}[n] + \frac{1}{2}\delta\mathbf{x}[n]^T V_{\mathbf{xx},n}\delta\mathbf{x}[n] = Q_n + \begin{bmatrix} Q_{\mathbf{x},n} \\ Q_{\mathbf{u},n} \end{bmatrix}^T \begin{bmatrix} \delta\mathbf{x}[n] \\ \delta\mathbf{u}[n]^* \end{bmatrix} + \frac{1}{2}\Big[$$

Compare terms in $(\cdot)\delta\mathbf{x}[n]$ and $1/2\delta\mathbf{x}[n]^T(\cdot)\delta\mathbf{x}[n]$, find $V_{\mathbf{x},n}$, and $V_{\mathbf{xx},n}$ and implement the corresponding function below.

IMPORTANT: Do not simplify the expression you obtain for V_x and V_{xx} by assuming that k and K have the form computed by the function `gains`.

In [10]:
```
def V_terms(Q_x, Q_u, Q_xx, Q_ux, Q_uu, K, k):
    # TODO: Implement V_x and V_xx, hint: use the A.dot(B) function for m
    V_x = Q_x + K.T @ Q_u + k.T @ Q_ux + K.T @ Q_uu @ k #np.zeros(Q_x.sha
    # print(Q_xx.shape, Q_ux.T.shape, K.shape, K.T.shape, Q_ux.shape, K.s
    V_xx = Q_xx + 2 * Q_ux.T @ K + K.T @ Q_uu @ K  #np.zeros(Q_xx.shape)

    return V_x, V_xx
```

Expected Cost Reduction

We can also estimate by how much we expect to reduce the cost by applying the

optimal controls. Simply subtract the previous nominal Q-value ($\delta\mathbf{x}[n] = 0$ and $\delta\mathbf{u}[n] = 0$) from the value function. The result is implemented below and is a useful aid in checking how accurate the quadratic approximation is during convergence of iLQR and adapting stepsize and regularization.

In [11]:
```
def expected_cost_reduction(Q_u, Q_uu, k):
    return -Q_u.T.dot(k) - 0.5 * k.T.dot(Q_uu.dot(k))
```

Forward Pass

We have now have all the ingredients to implement the forward pass and the backward pass of iLQR. In the forward pass, at each timestep the new updated control $\mathbf{u}' = \bar{\mathbf{u}} + k + K(x' - \bar{\mathbf{x}})$ is applied and the dynamis propagated based on the updated control. The nominal control and state trajectory $\bar{\mathbf{u}}, \bar{\mathbf{x}}$ with which we computed k and K are then updated and we receive a new set of state and control trajectories.

In [12]:
```
def forward_pass(x_trj, u_trj, k_trj, K_trj):
    x_trj_new = np.zeros(x_trj.shape)
    x_trj_new[0, :] = x_trj[0, :]
    u_trj_new = np.zeros(u_trj.shape)
    # TODO: Implement the forward pass here
    for n in range(u_trj.shape[0]):
        # Note, converting from deviation variable to actual value variab
        u_trj_new[n,:] = u_trj[n,:] + k_trj[n,:] + K_trj[n,:] @ (x_trj_ne
        x_trj_new[n+1,:] = discrete_dynamics(x_trj_new[n,:], u_trj_new[n,
    return x_trj_new, u_trj_new
```

Backward Pass

The backward pass starts from the terminal boundary condition $V(\mathbf{x}[N]) = \ell_f(\mathbf{x}[N])$, such that $V_{\mathbf{x},N} = \ell_{\mathbf{x},f}$ and $V_{\mathbf{xx},N} = \ell_{\mathbf{xx},f}$. In the backwards loop terms for the Q-function at n are computed based on the quadratic value function approximation at $n+1$ and the derivatives and hessians of dynamics and cost functions at n. To solve for the gains k and K an inversion of the matrix $Q_{\mathbf{uu}}$ is necessary. To ensure invertability and to improve conditioning we add a diagonal matrix to $Q_{\mathbf{uu}}$. This is equivalent to adding a quadratic penalty on the distance of the new control trajectory from the control trajectory of the previous iteration. The result is a smaller stepsize and more conservative convergence properties.

In [13]:
```
def backward_pass(x_trj, u_trj, regu):
    k_trj = np.zeros([u_trj.shape[0], u_trj.shape[1]])
    K_trj = np.zeros([u_trj.shape[0], u_trj.shape[1], x_trj.shape[1]])
    expected_cost_redu = 0
    # TODO: Set terminal boundary condition here (V_x, V_xx)
    V_x, V_xx = derivs.final(x_trj[-1])
    # print(x_trj.shape[1], u_trj.shape[1])
    # print(V_x.shape)
    # print(V_xx.shape)
    # V_x = np.zeros((x_trj.shape[1],))
    # V_xx = np.zeros((x_trj.shape[1], x_trj.shape[1]))
```

```
    for n in range(u_trj.shape[0] - 1, -1, -1):
        # TODO: First compute derivatives, then the Q-terms
        l_x, l_u, l_xx, l_ux, l_uu, f_x, f_u = derivs.stage(x_trj[n], u_t
        # print(l_ux.shape, f_u.shape, V_xx.shape, f_x.T.shape )
        Q_x, Q_u, Q_xx, Q_ux, Q_uu = Q_terms(l_x, l_u, l_xx, l_ux, l_uu,
        # Q_x = np.zeros((x_trj.shape[1],))
        # Q_u = np.zeros((u_trj.shape[1],))
        # Q_xx = np.zeros((x_trj.shape[1], x_trj.shape[1]))
        # Q_ux = np.zeros((u_trj.shape[1], x_trj.shape[1]))
        # Q_uu = np.zeros((u_trj.shape[1], u_trj.shape[1]))
        # We add regularization to ensure that Q_uu is invertible and nic
        Q_uu_regu = Q_uu + np.eye(Q_uu.shape[0]) * regu
        k, K = gains(Q_uu_regu, Q_u, Q_ux)
        k_trj[n, :] = k
        K_trj[n, :, :] = K
        V_x, V_xx = V_terms(Q_x, Q_u, Q_xx, Q_ux, Q_uu, K, k)
        expected_cost_redu += expected_cost_reduction(Q_u, Q_uu, k)
    return k_trj, K_trj, expected_cost_redu
```

Main Loop

The main iLQR loop consists of iteratively applying the forward and backward pass. The regularization is adapted based on whether the new control and state trajectories improved the cost. We lower the regularization if the total cost was reduced and accept the new trajectory pair. If the total cost did not decrease, the trajectory pair is rejected and the regularization is increased. You may want to test the algorithm with deactivated regularization and observe the changed behavior. The main loop stops if the maximum number of iterations is reached or the expected reduction is below a certain threshold.

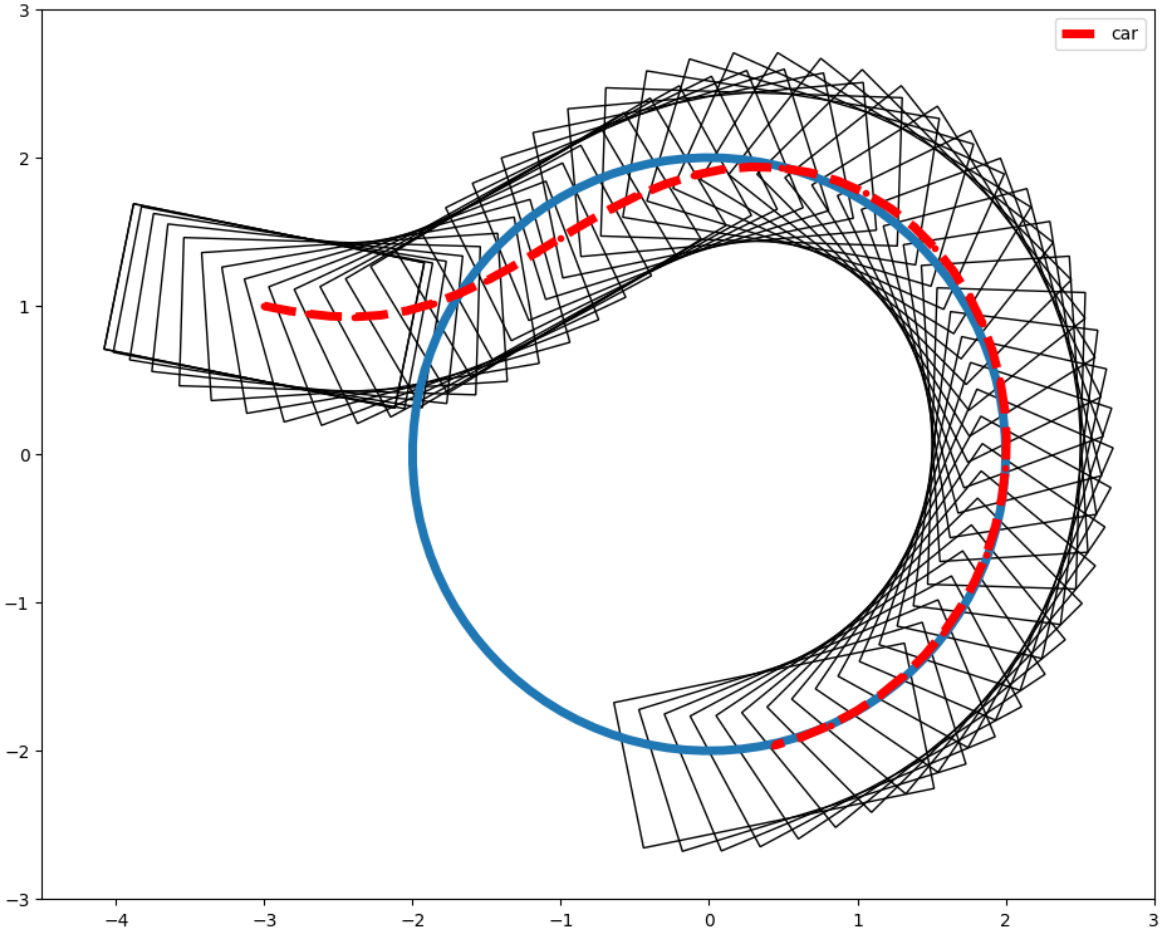
If you have correctly implemented all subparts of the iLQR you should see that the car plans to drive around the circle.

In [14]:
```
def run_ilqr(x0, N, max_iter=50, regu_init=100):
    # First forward rollout
    u_trj = np.random.randn(N - 1, n_u) * 0.0001
    x_trj = rollout(x0, u_trj)
    total_cost = cost_trj(x_trj, u_trj)
    regu = regu_init
    max_regu = 10000
    min_regu = 0.01

    # Setup traces
    cost_trace = [total_cost]
    expected_cost_redu_trace = []
    redu_ratio_trace = [1]
    redu_trace = []
    regu_trace = [regu]

    # Run main loop
    for it in range(max_iter):
        # Backward and forward pass
        k_trj, K_trj, expected_cost_redu = backward_pass(x_trj, u_trj, re
        x_trj_new, u_trj_new = forward_pass(x_trj, u_trj, k_trj, K_trj)
        # Evaluate new trajectory
        total_cost = cost_trj(x_trj_new, u_trj_new)
```

```
        cost_redu = cost_trace[-1] - total_cost
        redu_ratio = cost_redu / abs(expected_cost_redu)
        # Accept or reject iteration
        if cost_redu > 0:
            # Improvement! Accept new trajectories and lower regularizati
            redu_ratio_trace.append(redu_ratio)
            cost_trace.append(total_cost)
            x_trj = x_trj_new
            u_trj = u_trj_new
            regu *= 0.7
        else:
            # Reject new trajectories and increase regularization
            regu *= 2.0
            cost_trace.append(cost_trace[-1])
            redu_ratio_trace.append(0)
        regu = min(max(regu, min_regu), max_regu)
        regu_trace.append(regu)
        redu_trace.append(cost_redu)

        # Early termination if expected improvement is small
        if expected_cost_redu <= 1e-6:
            break

    return x_trj, u_trj, cost_trace, regu_trace, redu_ratio_trace, redu_t


# Setup problem and call iLQR
x0 = np.array([-3.0, 1.0, -0.2, 0.0, 0.0])
N = 50
max_iter = 50
regu_init = 100
x_trj, u_trj, cost_trace, regu_trace, redu_ratio_trace, redu_trace = run_
    x0, N, max_iter, regu_init
)


plt.figure(figsize=(9.5, 8))
# Plot circle
theta = np.linspace(0, 2 * np.pi, 100)
plt.plot(r * np.cos(theta), r * np.sin(theta), linewidth=5)
ax = plt.gca()

# Plot resulting trajecotry of car
plt.plot(x_trj[:, 0], x_trj[:, 1], 'r.--',linewidth=5, label='car')
w = 2.0
h = 1.0

# Plot rectangles
for n in range(x_trj.shape[0]):
    rect = mpl.patches.Rectangle((-w / 2, -h / 2), w, h, fill=False)
    t = (
        mpl.transforms.Affine2D()
        .rotate_deg_around(0, 0, np.rad2deg(x_trj[n, 2]))
        .translate(x_trj[n, 0], x_trj[n, 1])
        + ax.transData
    )
    rect.set_transform(t)
    ax.add_patch(rect)
ax.set_aspect(1)
plt.ylim((-3, 3))
```

```
plt.xlim((-4.5, 3))
plt.tight_layout()
plt.legend()
```

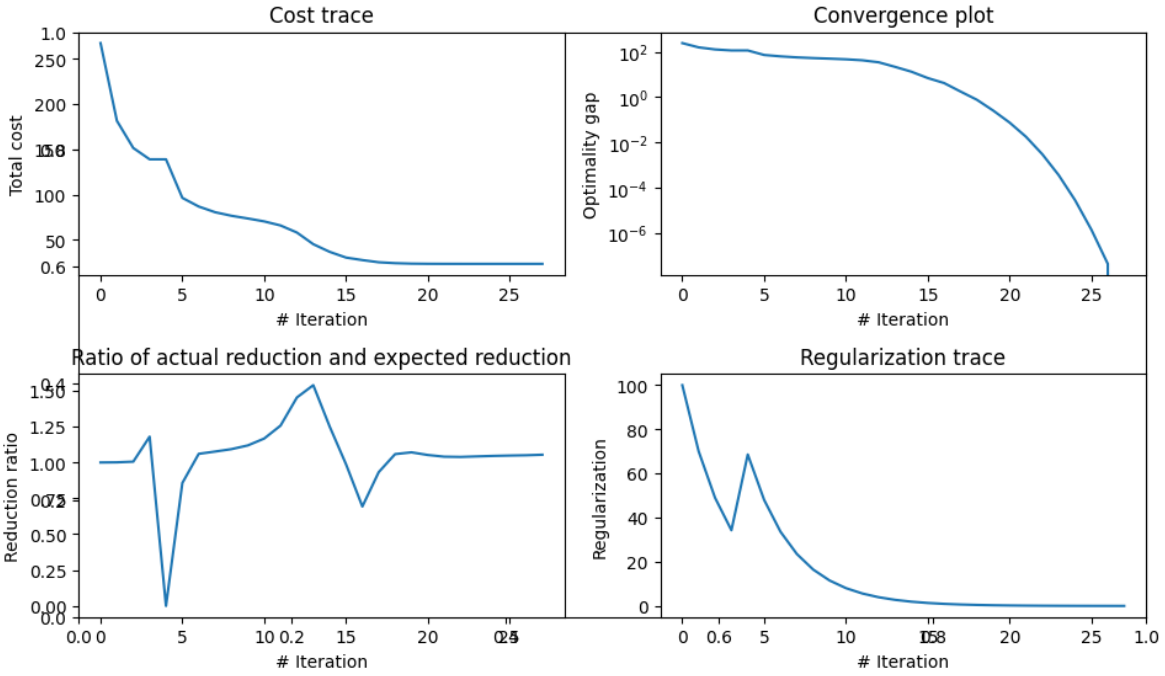
Out[14]: <matplotlib.legend.Legend at 0x76b43018aa20>



In [15]:
```
x_trj[:, 0], x_trj[:, 1]
```

Out[15]:
```
(array([-3.        , -3.        , -2.94083136, -2.83993574, -2.70940129,
        -2.5577048 , -2.39134197, -2.21590567, -2.0363648 , -1.85672521,
        -1.67959467, -1.50605512, -1.33587039, -1.16785456, -1.00025397,
        -0.83108744, -0.65844611, -0.48076947, -0.29710275, -0.10731818,
         0.08774262,  0.28625751,  0.48557553,  0.68251165,  0.87371035,
         1.05598263,  1.22654544,  1.38313959,  1.52404313,  1.64801567,
         1.7542098 ,  1.84207659,  1.91128163,  1.96163966,  1.99307012,
         2.00557241,  1.99921787,  1.97415494,  1.93062306,  1.86897153,
         1.78967931,  1.69337254,  1.58083721,  1.45302509,  1.31105233,
         1.15619112,  0.98985532,  0.81358219,  0.62901198,  0.4378674
8]),
 array([ 1.        ,  1.        ,  0.98800592,  0.96847036,  0.94711802,
         0.93064931,  0.92558794,  0.93702884,  0.96770651,  1.01776235,
         1.08524787,  1.16696187,  1.25917205,  1.35803588,  1.459759  ,
         1.56059935,  1.65681185,  1.74460263,  1.82014349,  1.87968124,
         1.91975133,  1.93746386,  1.93078791,  1.89874247,  1.84143121,
         1.75991923,  1.65600405,  1.53195348,  1.39026869,  1.23350394,
         1.06414921,  0.88456821,  0.69697767,  0.50345413,  0.30595609,
         0.10635242, -0.09354972, -0.29198183, -0.48719891, -0.67747316,
        -0.86109711, -1.03639485, -1.20173993, -1.35557732, -1.49644694,
        -1.62300627, -1.73404978, -1.82852411, -1.90553843, -1.9643707
]))
```

In [16]:
```
plt.subplots(figsize=(10, 6))
# Plot results
plt.subplot(2, 2, 1)
plt.plot(cost_trace)
plt.xlabel("# Iteration")
plt.ylabel("Total cost")
plt.title("Cost trace")

plt.subplot(2, 2, 2)
delta_opt = np.array(cost_trace) - cost_trace[-1]
plt.plot(delta_opt)
plt.yscale("log")
plt.xlabel("# Iteration")
plt.ylabel("Optimality gap")
plt.title("Convergence plot")

plt.subplot(2, 2, 3)
plt.plot(redu_ratio_trace)
plt.title("Ratio of actual reduction and expected reduction")
plt.ylabel("Reduction ratio")
plt.xlabel("# Iteration")

plt.subplot(2, 2, 4)
plt.plot(regu_trace)
plt.title("Regularization trace")
plt.ylabel("Regularization")
plt.xlabel("# Iteration")
plt.tight_layout()
```



Convergence Analysis

You can find some plots of the convergence traces captured throughout the iLQR solve process above. The convergence plot indicates that we have achieved superlinear convergence. In fact, iLQR achieves nearly second order convergence. In the case of linear convergence (e.g. gradient descent), the graph would show a line. While the integrated regularization improves robustness it damps convergence in the early

iteration steps.

In the ideal case, the expected reduction and the actual reduction should be the same, i.e. the reduction ratio remains around 1. If that is the case, the quadratic approximation of costs and linear approximation of the dynamics are very accurate. If the ratio becomes significantly lower than 1, the regularization needs to be increased and thus the stepsize reduced.

Autograding

You can check your work by running the following cell.

In [17]:
```
from underactuated.exercises.grader import Grader
from underactuated.exercises.trajopt.test_ilqr_driving import TestIlqrDri

Grader.grade_output([TestIlqrDriving], [locals()], "results.json")
Grader.print_test_results("results.json")
```

```
Total score is 18/18.

Score for Test discrete_dynamics is 1/1.

Score for Test rollout is 1/1.

Score for Test cost_trj is 1/1.

Score for Test Q_terms is 5/5.

Score for Test gains is 3/3.

Score for Test V_terms is 3/3.

Score for Test forward_pass is 2/2.

Score for Test backward_pass is 2/2.
```

In [18]:
```
# locals()
```