Spring 2022

Stanford
AA 203: Optimal and Learning-based Control
Homework #3, due May 24 by 11:59 pm
Pei-Chen Wu pcwu1023
Collaborator: Kevin Lee and Albert Chan

Learning goals for this problem set:

Problem 1: To gain experience with data-driven learning in a "real world" setting.

Problem 2: To study sequential convex programming in nonlinear trajectory optimization and build familiarity with CVXPY.

Problem 3: To gain familiarity with tools for HJ reachability and develop an understanding of sub-level sets in the context of backward reachability.

Problem 4: To understand the basics of feasibility in model predictive control.

Problem 5: To introduce algorithmic details in designing terminal ingredients for model predictive control.

Problem 1: Widget sales

You have just purchased Widget Co., a small shop selling widgets. Congratulations! Widget Co. is in the business of buying widgets wholesale, and selling them to consumers at a markup. While the previous owners were losing money, you suspect that you can apply your knowledge of optimal control and reinforcement learning to turn the business around.

The shop is able to store between 0 and 5 widgets at a time. We write the number of widgets held in the shop on day t as s_t. Every day, you choose how many widgets to order from your supplier. You can order either zero widgets, a "half order" of 2 widgets, or a "full order" of 4 widgets. We write the number of widgets ordered to arrive on day t as a_t. A random number of customers (following an unknown distribution, though this distribution may be assumed to be consistent across all days) come to Widget Co. every day; each customer buys a widget if there are any available. We write the demand on day t as d_t, and assume $d_t \leq 5$. At the end of each day, you record a net profit r_t for that day.

(a) The previous owners were nice enough to give you the details of their last three years of operation. In particular, they have provided a dataset $\mathcal{D} = \{(s_t, a_t, r_t)\}_{t=1}^{T}$ containing records for each day t of the last three years. In `starter_widget_sales.py`, implement a Q-learning algorithm to learn tabulated Q-values from this dataset. Provide only the code you add to the provided starter file.

CODE:

```
# Do Q-learning
γ = 0.95                    # discount factor
α = 1e-2                    # learning rate
num_epochs = 5 * int(1/α)   # number of epochs

Q = np.zeros((S.size, A.size))
Q_epoch = np.zeros((num_epochs + 1, S.size, A.size))

for k in tqdm(range(1, num_epochs + 1)):
    # Shuffle transition tuple indices
    shuffled_indices = rng.permutation(T)

    # ######################## PART (a): YOUR CODE BELOW ########################

    # INSTRUCTIONS: Update `Q` using Q-learning.
    action_map = {0:0, 2:1, 4:2}
    for t in shuffled_indices:
        s = int(log['s'][t])
        a = int(log['a'][t])
        u = action_map[a]
        r = log['r'][t]
        s_next = int(log['s'][t+1])

        Q[s,u] = Q[s,u] + α * (r + γ * np.max(Q[s_next]) - Q[s,u])

    # ############################ END PART (a) ############################

    # Record Q-values for this epoch
    Q_epoch[k] = Q
```

Figure 1: Code P1 (a)

While you were busy with your model-free learning, your modelling-inclined intern noticed that the dynamics of the number of widgets in the shop each day are described by

$$s_{t+1} = f(s_t, a_t, d_t) := \begin{cases} 0, & s_t + a_t - d_t < 0 \\ 5, & s_t + a_t - d_t > 5 \\ s_t + a_t - d_t, & \text{otherwise} \end{cases},$$

and the daily net profit is

$$r(s_t, a_t, d_t) = c_{\text{sell}} \min(s_t + a_t, d_t) - c_{\text{rent}} - c_{\text{storage}} s_t - g_{\text{order}}(a_t),$$

where $c_{\text{sell}} = 1.2$ is the price you set for each widget, $c_{\text{rent}} = 1$ is the fixed rent on your shop, $c_{\text{storage}} = 0.05$ is the cost for storing each widget overnight, and $g_{\text{order}}(a_t) = \sqrt{a_t}$ is the cost of ordering widgets from your supplier. The quantity $\min(s_t + a_t, d_t)$ is the "satisfied demand" on day t.

Moreover, after a few weeks of sales, your intern determined that the daily demand distribution for your widgets seems to be

$$d_t = \begin{cases} 0, & \text{with probability } 0.1 \\ 1, & \text{with probability } 0.3 \\ 2, & \text{with probability } 0.3 \\ 3, & \text{with probability } 0.2 \\ 4, & \text{with probability } 0.1 \end{cases}.$$

(b) In `starter_widget_sales.py`, implement value iteration to learn tabulated Q-values from the model your intern has provided. Submit only the code you add to the provided starter file. Also submit the plot generated by `starter_widget_sales.py` of Q-values from Q-learning compared to those from value iteration. What do you notice about the learned Q-values compared to those from value iteration? Why do you think this occurs?

ANSWER: The optimal policy are pretty similar except for the $s = 2$. The optimal policy we got from Value iteration is 0. The optimal policy we got from Q learning is 2. Q learning has higher values compared to the value iteration. Q learning is a model free method, the agent does not know the demand distribution. Instead, it considers discounted factor. The value iteration is learning the expected reward when you are given a state x, based on the problem statement, the expected reward will be a bit lower because demand probability distribution.

```python
# Do value iteration
converged = False
eps = 1e-4
max_iters = 500
Q_vi = np.zeros((S.size, A.size))
Q_vi_prev = np.full(Q_vi.shape, np.inf)

for k in tqdm(range(max_iters)):

    # ######################## PART (b): YOUR CODE BELOW ########################

    # INSTRUCTIONS: Update `Q_vi` using value iteration.
    action_map = {0:0, 2:1, 4:2}
    for s in S:
        for a in A:
            r = 0
            for d in D:
                p = P[d]
                s_next = transition(s,a,d)
                r += p * reward(s,a,d) + γ * p * np.max(Q_vi[s_next])
            Q_vi[s,action_map[a]] = r

    # ############################## END PART (b) ##############################

    if np.max(np.abs(Q_vi - Q_vi_prev)) < eps:
        converged = True
        print('Value iteration converged after {} iterations.'.format(k))
        break
    else:
        np.copyto(Q_vi_prev, Q_vi)

if not converged:
    raise RuntimeError('Value iteration did not converge!')
```

Figure 2: Code P1 (b)

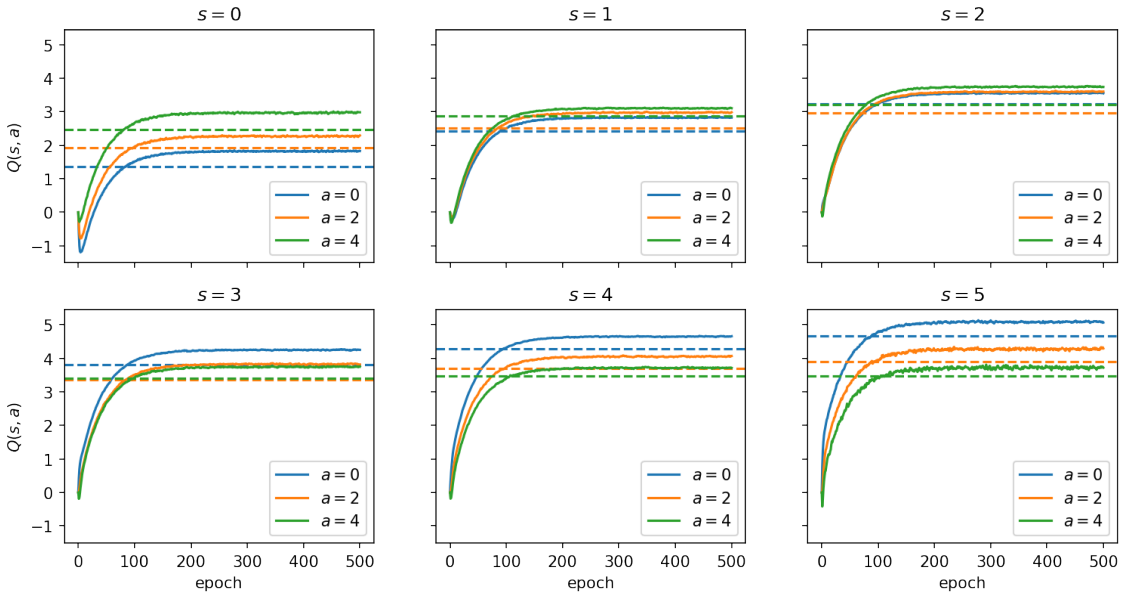


Figure 3: QL and VI P1 (b)

(c) In `starter_widget_sales.py`, compute an optimal policy $\pi^*_{\text{QL}}(s_t)$ based on your Q-learning work, and another optimal policy $\pi^*_{\text{VI}}(s_t)$ based on your value iteration work. Report each optimal policy, simulate each one over five years, and compute the cumulative profit $\sum_{k=0}^{t} r_k$ for each day t and for each optimal policy. Submit only the code you add to the provided starter file. Also submit the plot generated by `starter_widget_sales.py` comparing the cumulative profits over time. What do you notice about the difference between the two cumulative profit trends? Why do you think this occurs?

ANSWER: The Q learning method could generate higher cumulative reward. The reason might be one of Q learning's optimal policy is different from value iteration. While $s = 2$, Q learning suggests that we order 4 widgets, however, value iteration suggests that we order nothing. This could make difference on the profit in the long run. From demand distribution, $d_t = 1, 2$ has higher probability to occur, that's why it will be better to always have at least 2 widgets in the storage.

```
Optimal policy (Q-learning):      [2 2 2 0 0 0]
Optimal policy (value iteration): [2 2 0 0 0 0]
```

Figure 4: optimal Policy P1 (c)

```
# ######################### PART (c): YOUR CODE BELOW ########################

# INSTRUCTIONS: Compute the optimal actions `a_opt_ql` and `a_opt_vi` using the
#               Q-values from Q-learning and value iteration, respectively.
#               Both `a_opt_ql` and `a_opt_vi` should be `np.ndarray`s, where
#               each entry is the optimal action for the corresponding state.
#
#               Also, simulate each optimal policy and compute the history of
#               cumulative profits `profit_ql` and `profit_vi` over 5 years
#               (at 365 days per year).

T = 5 * 365

# TODO: replace the next four lines with your code
a_opt_ql = np.argmax(Q, axis=1)
profit_ql = np.cumsum(simulate(rng, lambda s, A=A: A[a_opt_ql[int(s)]], T)[2])
a_opt_vi = np.argmax(Q_vi, axis=1)
profit_vi = np.cumsum(simulate(rng, lambda s, A=A: A[a_opt_vi[int(s)]], T)[2])

# ############################## END PART (c) ##############################
```

Figure 5: Code P1 (c)

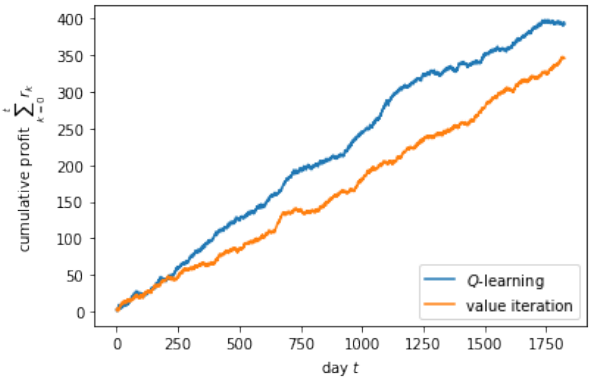


Figure 6: Cumulative Profit (c)

Problem 2: Cart-pole swing-up with limited actuation

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. We will solve this constrained optimal control problem with a direct method. The transcribed optimal control problem we would like to solve is

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

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

However, this optimal control problem is non-convex due to the nonlinear equality constraints $s_{k+1} = f_d(s_k, u_k)$. We will circumvent this issue with Sequential Convex Programming (SCP). Recall that the key idea of SCP is to iteratively re-linearize the dynamics around a nominal trajectory and solve a convex approximation of the optimal control problem near this trajectory. Since linearization provides a good approximation to the nonlinear dynamics only in a small neighborhood around the nominal trajectory $(\bar{s}, \bar{u})$, the accuracy of the convex model may be poor if (s, u) deviates far from $(\bar{s}, \bar{u})$. 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 $(\bar{s}, \bar{u})$ described by

$$\|s - \bar{s}\|_\infty \leq \rho, \quad \|u - \bar{u}\|_\infty \leq \rho,$$

for some constant $\rho > 0$. Moreover, the terminal constraint $s_N = s^*$ 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 $s_N = s^*$ and add the terminal state cost term $(s_N - s^*)^\mathsf{T} P (s_N - s^*)$ as a proxy, where $P \succ 0$ has large entries.

In `starter_cartpole_swingup_limited_actuation.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 in `starter_cartpole_swingup_limited_actuation.py` before you continue. Only submit code that you are asked to add to the provided starter file and plots.

(a) Derive the convex approximation of the nonlinear optimal control problem around a given nominal trajectory $(\bar{s}_{0:N}, \bar{u}_{0:N-1})$, including the control constraint set $\mathcal{U} \coloneqq [-r_u, r_u]$ for some $r_u > 0$, 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 $(\bar{s}_{0:N}, \bar{u}_{0:N-1})$. Derive A_k, B_k, and c_k in terms of $\bar{s}_k$, $\bar{u}_k$, and f_d.

$$s_{k+1} = f_d(s_k, u_k) = f_d(\bar{s}_k, \bar{u}_k) + \nabla_s f_d(\bar{s}_k, \bar{u}_k)(s_k - \bar{s}_k) + \nabla_u f_d(\bar{s}_k, \bar{u}_k)(u_k - \bar{u}_k)$$

$$f_d(s_k, u_k) = f_d(\bar{s}_k, \bar{u}_k) + \nabla_s f_d(\bar{s}_k, \bar{u}_k) s_k - \nabla_s f_d(\bar{s}_k, \bar{u}_k) \bar{s}_k + \nabla_u f_d(\bar{s}_k, \bar{u}_k) u_k - \nabla_u f_d(\bar{s}_k, \bar{u}_k) \bar{u}_k$$

$$A_k = \nabla_s f_d(\bar{s}_k, \bar{u}_k)$$

$$B_k = \nabla_u f_d(\bar{s}_k, \bar{u}_k)$$

$$c_k = f_d(\bar{s}_k, \bar{u}_k) - \nabla_s f_d(\bar{s}_k, \bar{u}_k) \bar{s}_k - \nabla_u f_d(\bar{s}_k, \bar{u}_k) \bar{u}_k$$

(b) Use JAX to complete the function `linearize` such that it computes A_k, B_k, and c_k from $\bar{s}_k$, $\bar{u}_k$, and f_d.

```
@partial(jax.jit, static_argnums=(0,))
@partial(jax.vmap, in_axes=(None, 0, 0))
def linearize(fd: callable,
              s: jnp.ndarray,
              u: jnp.ndarray):
    """Linearize the function `fd(s,u)` around `(s,u)`."""
    # ######################## PART (b): YOUR CODE BELOW ######################

    # INSTRUCTIONS: Use JAX to linearize `fd` around `(s,u)`.

    # TODO: Replace the four lines below with your code.
    #n, m = s.size, u.size
    A = jax.jacfwd(fd,0)(s,u)
    B = jax.jacfwd(fd,1)(s,u)
    c = fd(s,u) - A@s - B@u

    # ############################## END PART (b) ##############################

    return A, B, c
```

Figure 7: P2 Code (a)

(c) Complete the function `scp_iteration`. Specifically, given a nominal trajectory $(\bar{s}_{0:N}, \bar{u}_{0:N-1})$, use CVXPY to specify and solve the convex optimization problem derived in part (a) to obtain an updated solution $(s_{0:N}, u_{0:N-1})$.

```python
def scp_iteration(fd: callable, P: np.ndarray, Q: np.ndarray, R: np.ndarray,
                  N: int, s_bar: np.ndarray, u_bar: np.ndarray,
                  s_goal: np.ndarray, s0: np.ndarray,
                  ru: float, ρ: float):
    """Solve a single SCP sub-problem for the cart-pole swing-up problem."""
    A, B, c = linearize(fd, s_bar[:-1], u_bar)
    A, B, c = np.array(A), np.array(B), np.array(c)
    n = Q.shape[0]
    m = R.shape[0]
    s_cvx = cvx.Variable((N + 1, n))
    u_cvx = cvx.Variable((N, m))

    # ###################### PART (c): YOUR CODE BELOW ######################

    # INSTRUCTIONS: Construct and solve the convex sub-problem for SCP.

    # TODO: Replace the two lines below with your code.
    objective = cvx.quad_form(s_cvx[N]-s_goal,P)
    constraints = [s_cvx[0] == s0]
    for k in range(N):
        objective += cvx.quad_form(s_cvx[k]-s_goal,Q) + cvx.quad_form(u_cvx[k],R)
        constraints += [cvx.norm(s_cvx[k] - s_bar[k], "inf") <= ρ, cvx.norm(u_cvx[k] - u_bar[k], "inf") <= ρ]
        constraints += [u_cvx[k] <= ru, u_cvx[k] >= -ru]
        constraints += [s_cvx[k+1] == A[k,:,:]@s_cvx[k] + B[k,:,:]@u_cvx[k] + c[k,:]]

    # ########################### END PART (c) ###########################

    prob = cvx.Problem(cvx.Minimize(objective), constraints)
    prob.solve()

    if prob.status != 'optimal':
        raise RuntimeError('SCP solve failed. Problem status: ' + prob.status)

    s = s_cvx.value
    u = u_cvx.value
    obj = prob.objective.value

    return s, u, obj
```

Figure 8: P2 Code (c)

(d) Run `starter_cartpole_swingup_limited_actuation.py`. Submit the generated state and control trajectory 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 remain upright and keep it there. In words, suggest a control scheme to do this.
ANSWER: Once the cart-pole reaches upright state, we can apply LQR that we developed from last HW, linearized around upright state, to keep the pendulum up.

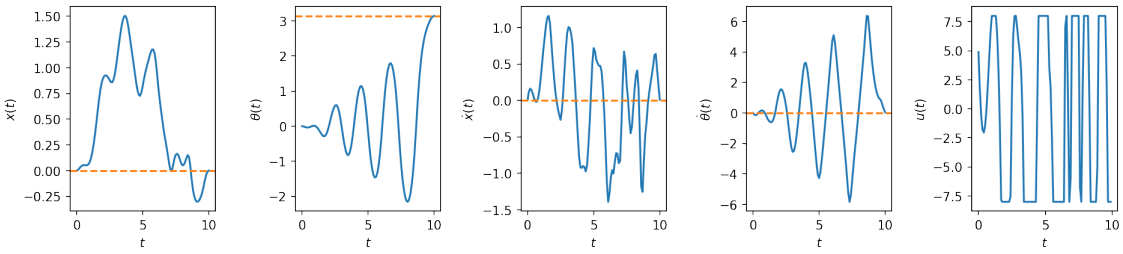


Figure 9: P2 State and Control Trajectory (d)

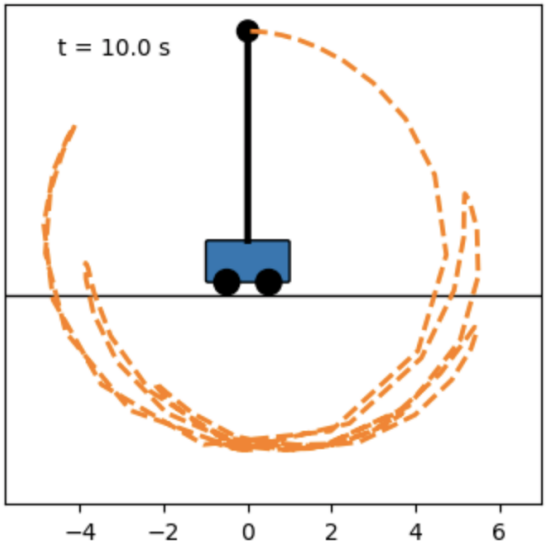


Figure 10: P2 Cart-pole Animation (d)

Problem 3: Hamilton-Jacobi reachability

Consider the goal of developing a self-righting quadrotor, i.e., a flying drone that you can throw into the air at various poses and velocities which will autonomously regulate itself to level flight while obeying dynamics, control, and operational-envelope constraints. For this problem, we consider the 6-D dynamics of a planar quadrotor described by

$$\begin{bmatrix} \dot{x} \\ \dot{v}_x \\ \dot{y} \\ \dot{v}_y \\ \dot{\phi} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} v_x \\ \frac{-(T_1+T_2)\sin\phi - C_D^v v_x}{m} \\ v_y \\ \frac{(T_1+T_2)\cos\phi - C_D^v v_y}{m} - g \\ \omega \\ \frac{(T_2-T_1)\ell - C_D^\phi \omega}{I_{yy}} \end{bmatrix}, \quad T_1, T_2 \in [0, T_{\max}], \tag{1}$$

where the state is given by the position in the vertical plane (x, y), translational velocity (v_x, v_y), pitch ϕ, and pitch rate ω; the controls are the thrusts (T_1, T_2) for the left and right prop respectively. Additional constants appearing in the dynamics above are gravitational acceleration g, the quadrotor's mass m, moment of inertia (about the out-of-plane axis) I_{yy}, half-length ℓ, and translational and rotational drag coefficients C_D^v and C_D^ϕ, respectively (see `starter_hj_reachability.py` for precise values of these constants in `PlanarQuadrotor.__init__`).

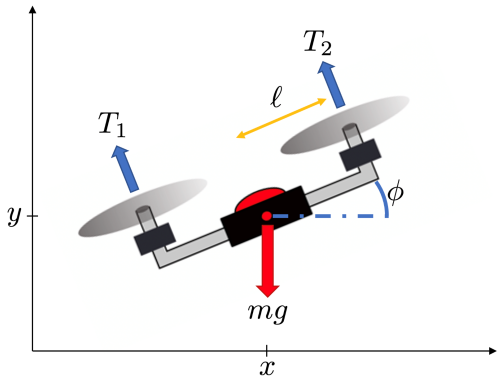


Figure 11: A planar quadrotor.

We will approach the problem of self-righting through continuous-time dynamic programming, specifically a Hamilton-Jacobi-Bellman (HJB) formulation.[1] To help mitigate the curse of dimensionality, we ignore the lateral motion (irrelevant to achieving level flight) and consider reduced 4-D dynamics with state vector $\mathbf{x} := (y, v_y, \phi, \omega) \in \mathbb{R}^4$. For these reduced dynamics, we define the target set

$$\mathcal{T} = [3, 7] \times [-1, 1] \times [-\pi/12, \pi/12] \times [-1, 1] \subset \mathbb{R}^4.$$

We assume that once the planar quadrotor reaches this set, we have another controller (e.g., an LQR controller linearized around hover) that can take over to maintain level flight.

To bound the domain of our dynamic programming problem (and also to ensure that our quadrotor doesn't plow into the ground), in addition to the dynamics and control constraints given in (1) we

[1] One might also consider an HJI-based extension to handle worst-case disturbances (e.g., wind), but for simplicity in this exercise we just consider the undisturbed dynamics.

would also like to constrain our planar quadrotor to stay within the operational envelope

$$\mathcal{E} = [1, 9] \times [-6, 6] \times [-\infty, \infty] \times [-8, 8].$$

Reaching the target set $\mathcal{T}$ while *avoiding* the obstacle set $\mathcal{E}^\complement$ (i.e., the set complement of $\mathcal{E}$) is referred to as a *reach-avoid* problem. If we can construct two real-valued, Lipschitz continuous functions $h(\mathbf{x}), e(\mathbf{x})$ defined over the state domain such that

$$\mathbf{x} \in \mathcal{T} \iff h(\mathbf{x}) \leq 0, \quad \mathbf{x} \in \mathcal{E} \iff e(\mathbf{x}) \leq 0,$$

i.e., $\mathcal{T}, \mathcal{E}$ are the zero-sublevel sets of h, e respectively, then it may be shown (see, e.g., [FCTS15, Theorem 1]) that the value function $V(\mathbf{x}, t)$ defined as

$$\begin{aligned} V(\mathbf{x}_0, t_0) = \min_{\mathbf{u}(\cdot)} \min_{\tau \in [t_0, 0]} & h(\mathbf{x}(\tau)) \\ \text{s.t. } & \dot{\mathbf{x}}(\tau) = f(\mathbf{x}(\tau), \mathbf{u}(\tau)) \quad \forall \tau \in [t_0, 0] \\ & \mathbf{x}(\tau) \in \mathcal{E} \quad \forall \tau \in [t_0, 0] \\ & \mathbf{x}(t_0) = \mathbf{x}_0 \end{aligned}$$

(where f is the relevant portion of the full dynamics (1)) satisfies the HJB PDE[2]

$$\begin{aligned} \max \left\{ \frac{\partial V}{\partial t}(\mathbf{x}, t) + \min\{0, H(\mathbf{x}, \nabla_{\mathbf{x}} V(\mathbf{x}, t))\}, e(\mathbf{x}) - V(\mathbf{x}, t) \right\} = 0 \\ \text{where } H(\mathbf{x}, \mathbf{p}) = \min_{\mathbf{u}} \mathbf{p}^T f(\mathbf{x}, \mathbf{u}), \\ V(\mathbf{x}, 0) = \max\{h(\mathbf{x}), e(\mathbf{x})\}. \end{aligned}$$

Implementing an appropriate solver for this type of PDE is somewhat nontrivial (see, e.g., [Mit02] for details); for this exercise we will use an existing solver – you will be responsible for setting the problem up and interpreting the results.

If you are running your code locally on your own machine, install the solver at the command line using `pip` via the command:

```
pip install --upgrade hj-reachability
```

Otherwise, if you are using Google Colab, run a cell containing:

```
!pip install --upgrade hj-reachability
```

For this problem, you will fill parts of `starter_hj_reachability.py` in with your own code. When submitting code, only provide the methods or functions that you have been asked to modify.

(a) Subject to the control constraints $T_1, T_2 \in [0, T_{\max}]$, derive the locally optimal action that minimizes the Hamiltonian, i.e., for arbitrary $\mathbf{x}, \mathbf{p}$ compute

$$\mathbf{u}^* = \arg\min_{\mathbf{u}} \mathbf{p}^T f(\mathbf{x}, \mathbf{u}),$$

[2] This is similar to the backward reachable tube HJI PDE mentioned in class (omitting the disturbance), where as before the inner min ensures the value function is nondecreasing in time (so that as BRT computation proceeds backward in time, the value function is nonincreasing at successive iterations, i.e., you get to "lock in" the lowest value you ever achieve). The outer max is the new addition in this formulation compared to what we saw in class, and may be interpreted as always making sure $V(\mathbf{x}, t) \geq e(\mathbf{x})$ so that if $e(\mathbf{x}) > 0$ (i.e., the state is outside of the operating envelope) then also $V(\mathbf{x}, t) > 0$ (i.e., the state is outside the BRT of states that can reach the target collision-free).

where f denotes the last four rows of the dynamics defined by (1). Use this knowledge to implement the method `PlanarQuadrotor.optimal_control`.

ANSWER:

$$H(\mathbf{x}, \mathbf{p}) = \min_{\mathbf{u}} \mathbf{p}^T f(\mathbf{x}, \mathbf{u})$$

We can simplify the equation to the following form, since we only care about the terms that will be affected by control inputs at the moment.
$c1 = \frac{P_2 \cos\phi}{m}$ $c2 = \frac{P_4 l}{I_{yy}}$

$$H(\mathbf{x}, \mathbf{p}) = \min_{\mathbf{u}} \mathbf{p}^T f(\mathbf{x}, \mathbf{u}) = \min_{\mathbf{u}} \left\{ c1(T_1 + T_2) + c2(T_2 - T_1) \right\}$$

$$H(\mathbf{x}, \mathbf{p}) = \min_{\mathbf{u}} \mathbf{p}^T f(\mathbf{x}, \mathbf{u}) = \min_{\mathbf{u}} \left\{ T_1(c_1 - c_2) + T_2(c_1 + c_2) \right\}$$

The optimal control u^* will be,
$T_1 = 0$ if $C_1 - C_2 > 0$
$T_1 = T_{max}$ if $C_1 - C_2 < 0$
$T_2 = 0$ if $C_1 + C_2 > 0$
$T_2 = T_{max}$ if $C_1 + C_2 < 0$

```python
def optimal_control(self, state, grad_value):
    """Computes the optimal control realized by the HJ PDE Hamiltonian.

    Args:
        state: An unbatched (!) state vector, an array of shape `(4,)` containing `[y, v_y, phi, omega]`.
        grad_value: An array of shape `(4,)` containing the gradient of the value function at `state`.

    Returns:
        A vector of optimal controls, an array of shape `(2,)` containing `[T_1, T_2]`, that minimizes
        `grad_value @ self.dynamics(state, control)`.
    """
    # PART (a): WRITE YOUR CODE BELOW ###################################################
    # You may find `jnp.where` to be useful; see corresponding numpy docstring:
    # https://numpy.org/doc/stable/reference/generated/numpy.where.html
    y, dy, phi, dphi = state
    p1, p2, p3, p4 = grad_value

    c1 = p2*jnp.cos(phi)/self.m
    c2 = p4*self.l/self.Iyy

    Tmax = self.max_thrust_per_prop
    Tmin = self.min_thrust_per_prop


    T1 = jnp.where(c1-c2>0, Tmin, Tmax)
    T2 = jnp.where(c1+c2>0, Tmin, Tmax)

    return jnp.array([T1,T2])
    ######################################################################################
```

Figure 12: P3 code (a)

(b) Write down a functional form for $h(\mathbf{x})$ such that $\mathbf{x} \in \mathcal{T} \iff h(\mathbf{x}) \leq 0$. Implement the function `target_set`.

Hint: Note that $a(\mathbf{x}) \leq 0 \ \wedge \ b(\mathbf{x}) \leq 0 \iff \max\{a(\mathbf{x}), b(\mathbf{x})\} \leq 0$. This means that if you have multiple constraints represented as the zero-sublevel sets of multiple functions, then the conjunction of the constraints may be represented as a pointwise maximum of the functions.

$$\min\left\{|y-5|-2, |\dot{y}|-1, |\phi| - \frac{\pi}{12}, |\dot{\phi}|-1\right\} \tag{2}$$

```python
def target_set(state):
    """A real-valued function such that the zero-sublevel set is the target set.

    Args:
        state: An unbatched (!) state vector, an array of shape `(4,)` containing `[y, v_y, phi, omega]`.

    Returns:
        A scalar, nonpositive iff the state is in the target set.
    """
    # PART (b): WRITE YOUR CODE BELOW ###########################################
    y, dy, phi, dphi = state
    return jnp.max(jnp.array([jnp.abs(y-5)-2,jnp.abs(dy-0)-1,jnp.abs(phi-0)-jnp.pi/12,jnp.abs(dphi-0)-1]))
    ############################################################################
```

Figure 13: P3 code (b)

(c) Write down a functional form for $e(\mathbf{x})$ such that $\mathbf{x} \in \mathcal{E} \iff e(\mathbf{x}) \leq 0$. Implement the function `envelope_set`.

$$\min\left\{|y-5|-4, |\dot{y}|-6, |\dot{\phi}|-8\right\} \tag{3}$$

```python
def envelope_set(state):
    """A real-valued function such that the zero-sublevel set is the operational envelope.

    Args:
        state: An unbatched (!) state vector, an array of shape `(4,)` containing `[y, v_y, phi, omega]`.

    Returns:
        A scalar, nonpositive iff the state is in the operational envelope.
    """
    # PART (c): WRITE YOUR CODE BELOW ###########################################
    y, dy, phi, dphi = state
    return jnp.max(jnp.array([jnp.abs(y-5)-4,jnp.abs(dy-0)-6,jnp.abs(dphi-0)-8]))
    ############################################################################
```

Figure 14: P3 code (c)

(d) Run the rest of the script/cells to compute $V(\mathbf{x}, -5)$ and take a look at some of the controlled trajectories; hopefully they look reasonable (see the note below if you're picky/have extra time, though if the quad rights itself and gets to the target set $\mathcal{T}$ that's sufficient for our purposes). Do not submit any trajectory plots; instead include a 3D plot of the zero isosurface (equivalent of a contour/isoline, but in 3D) for a slice of the value function at some fixed y value (e.g., $y = 7.5$ as pre-selected in the starter code). Explain why one of the bumps/ridges (e.g., as highlighted by the red or blue arrow in Figure 15, which you may also use to check your work) has the shape that it does.

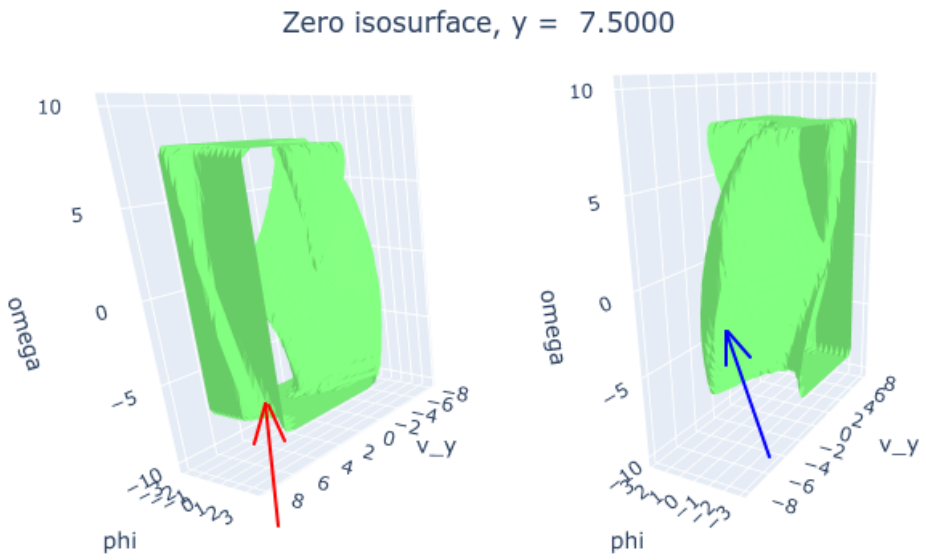


Figure 15: Example zero isosurface views. Can you explain why the red valley (outside the isosurface, i.e., unrecoverable initial conditions) or blue ridge (inside the isosurface, i.e., initial conditions that can reach the target collision-free) exhibit the "tilt" they have by considering the corresponding states?

Note: If the behavior of your control policy isn't as nice as you'd like (e.g., height/pitch oscillations), consider modifying your target set function $h(\mathbf{x})$ (e.g., by scaling how you account for each dimension in your construction). For the purpose of reachable set computation, at least theoretically[3] the zero-sublevel set of the value function V (corresponding to the set of feasible initial states) is unaffected by the details of h as long as $h(\mathbf{x}) \leq 0 \iff \mathbf{x} \in \mathcal{T}$. In the context of dynamic programming to compute an optimal control policy, however, $h(\mathbf{x})$ also defines the terminal cost in a way that materially affects the policy once the set is reached (though in practice, this is where we'd have some other stabilizing controller take over).

ANSWER: The isotropic surface of HJB is the nonlinear partial differential equation in the value function. We can see from the top view, if the drone velocity is positive and high, it can tilt and spin to recover to the good state. If it's dropping down too fast, it has to stay straight up to be able to recover. If along the lines of the initial fall velocity being low enough that the drone can recover from having an initial tilt and spin. If the initial velocity is too

[3]With a relatively coarse grid discretization and not-particularly-high-accuracy finite difference schemes/time integrators for PDE solving (sacrifices made so you don't have to wait for hours to see results), for numerical reasons the BRT may have some dependence on your formulation of $h(\mathbf{x})$.

high you get the concave bump and it becomes unrecoverable.

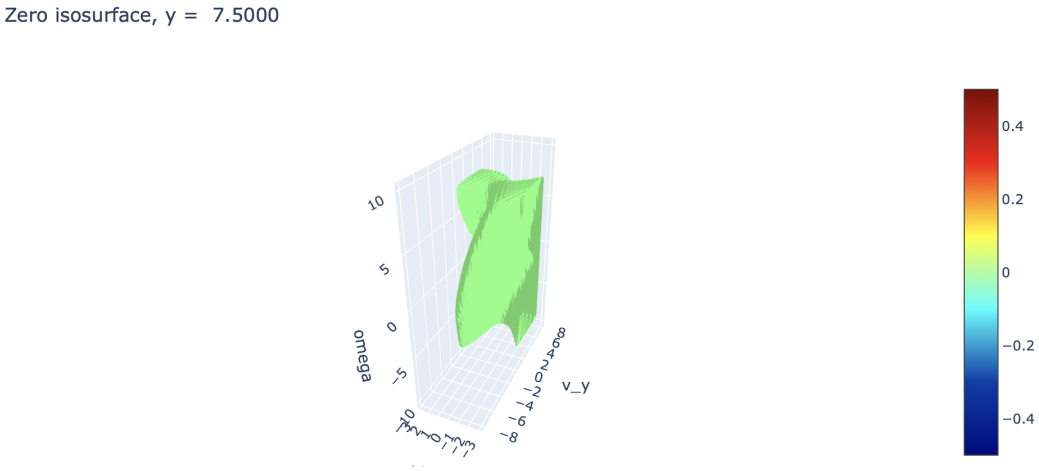


Figure 16: P3 Bumps (d)

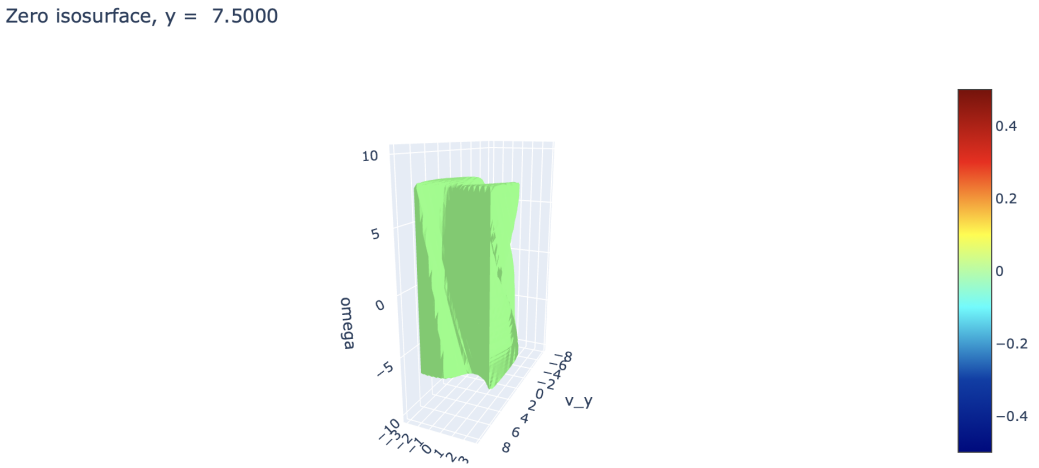


Figure 17: P3 Ridges (d)

(e) In a few sentences, write down some pros/cons of this approach (i.e., computing a policy using dynamic programming) for a self-righting quadrotor vs. alternatives, e.g., applying model-predictive control. Potential things to think/write about: computational resources (time, memory) required for online operation, local/global optimality, flexibility to accommodate additional obstacles in the environment, bang-bang controls, etc.

ANSWER: If comparing HJB to other methods, it might be hard to explicitly solve for T_1 and T_2 vs linearization. HJB is global and gives the feedback law for every initial condition once the value function has been computed. On the other hand, MPC is faster, but gives an approximate feedback control just for a single initial condition. HJB could return global optimality, which MPC might not provide global optimality, it only provide the optimal control at the time step within the sliding window. For obstacle avoidance, MPC might be a better choice, since it's computing faster for a single initial condition.

Problem 4: MPC feasibility

Consider the discrete-time LTI system

$$x_{k+1} = Ax_k + Bu_k.$$

We want to compute a receding horizon controller for a quadratic cost function, i.e.,

$$J(x_{0:N}, u_{0:N-1}) = x_N^\mathsf{T} P x_N + \sum_{k=0}^{N-1} \Big(x_k^\mathsf{T} Q x_k + u_k^\mathsf{T} R u_k \Big),$$

where $P, Q, R \succ 0$ are weight matrices. We must satisfy the state and input constraints $\|x_k\|_\infty \leq r_x$ and $\|u_k\|_\infty \leq r_u$, respectively. Also, we will enforce the terminal state constraint $\|x_N\|_\infty \leq r_f$, where we will tune $r_f \geq 0$. For $r_f = 0$, the terminal state constraint is equivalent to $x_N = 0$, while for $r_f \geq r_x$ we are just left with the original state constraint $\|x_N\|_\infty \leq r_x$.

For this problem, you will work with the starter code in `starter_mpc_feasibility.py`. Carefully review *all* of the code in this file before you continue. Only submit code you add and any plots that are generated by the file.

(a) Implement a receding horizon controller for this system using CVXPY in the function `do_mpc`. Run the remaining code to simulate closed-loop trajectories with $r_f \geq r_x$ from two different initial states, each with either $P = I$ or P as the unique positive-definite solution to the discrete algebraic Riccati equation (DARE)

$$A^\mathsf{T} P A - P - A^\mathsf{T} P B (R + B^\mathsf{T} P B)^{-1} B^\mathsf{T} P A + Q = 0.$$

Submit your code and the plot that is generated, which displays both the realized closed-loop trajectories and the predicted open-loop trajectories at each time step. Discuss your observations of any differences between the trajectories for the different initial conditions and values of P.

ANSWER: If DARE is used, the stationary solution of the forward pass is fully differentiable. Therefore, it will provide infinite-horizon optimality and stability, which from the result performs better than identity matrix.

```python
def do_mpc(x0: np.ndarray, A: np.ndarray, B: np.ndarray,
           P: np.ndarray, Q: np.ndarray, R: np.ndarray,
           N: int, rx: float, ru: float,
           rf: float):
    """Solve the MPC problem starting at state `x0`."""
    n, m = Q.shape[0], R.shape[0]
    x_cvx = cvx.Variable((N + 1, n))
    u_cvx = cvx.Variable((N, m))

    # ###################### PART (a): YOUR CODE BELOW ######################

    # INSTRUCTIONS: Construct and solve the MPC problem using CVXPY.

    # TODO: Replace the two lines below with your code.
    cost = cvx.quad_form(x_cvx[N],P)
    constraints = [x_cvx[0] == x0]
    constraints += [cvx.norm(x_cvx[N], "inf") <= rf]

    for k in range(N):
        cost += cvx.quad_form(x_cvx[k],Q) + cvx.quad_form(u_cvx[k],R)
        constraints += [cvx.norm(x_cvx[k], "inf") <= rx, cvx.norm(u_cvx[k], "inf") <= ru]
        constraints += [x_cvx[k+1] == A@x_cvx[k] + B@u_cvx[k]]
    constraints += [cvx.norm(x_cvx[N], "inf") <= rx]

    # ############################ END PART (a) ############################

    prob = cvx.Problem(cvx.Minimize(cost), constraints)
    prob.solve()
    x = x_cvx.value
    u = u_cvx.value
    status = prob.status

    return x, u, status
```

Figure 18: P4 code (a)

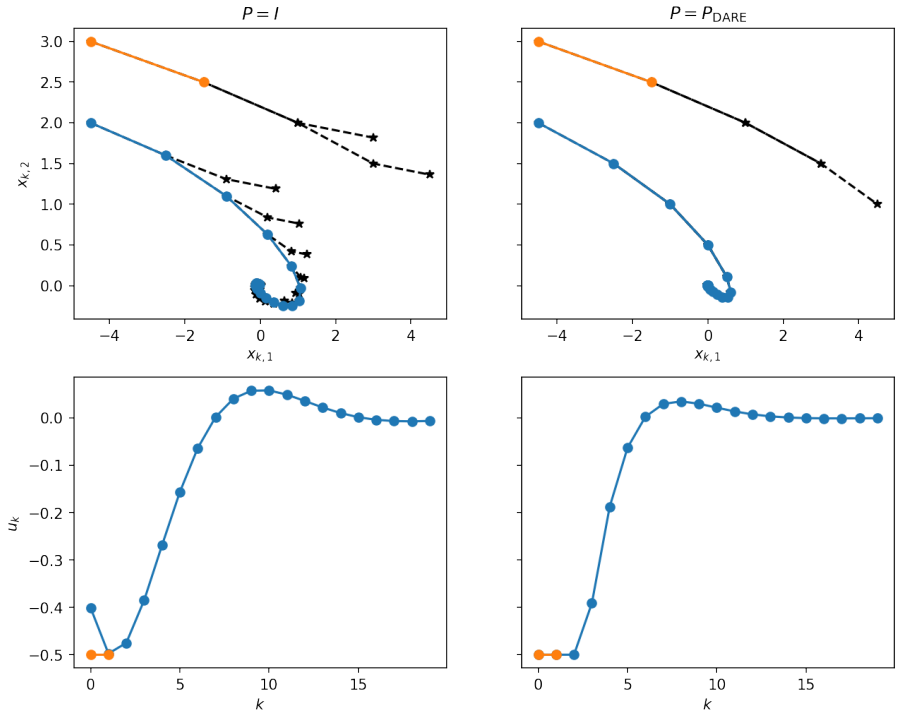


Figure 19: P4 MPC Feasibility Sim (a)

(b) Finish the function `compute_roa`, which computes the region of attraction (ROA) for fixed $P \succ 0$ and different values of N and r_f. Submit your code and the plot of the different ROAs. Compare and discuss your observations of the ROAs.

Hint: While debugging your code, you can set a small `grid_dim` to reduce the amount of time it takes to compute the ROAs. See `??` for the ROAs with `grid_dim = 5`. However, you must submit your plot of the ROAs with at least `grid_dim = 30`.

ANSWER: We can easily observe that if r_f is larger, the ROA region is bigger same as having higher N.

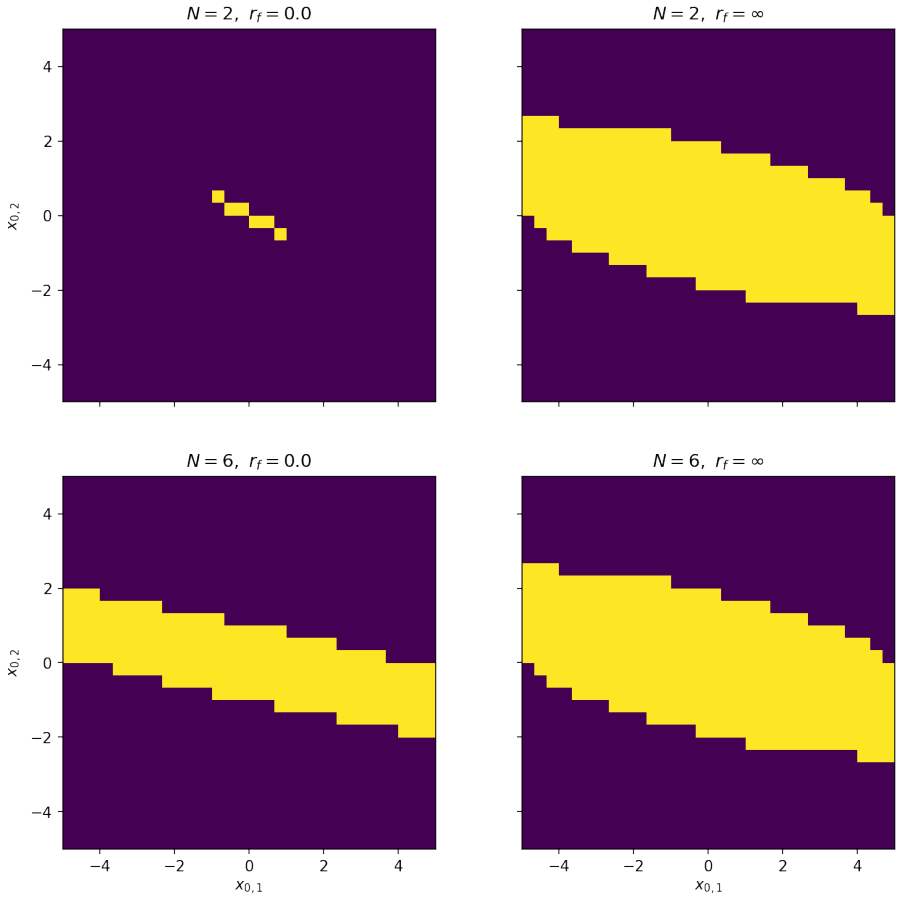


Figure 20: ROAs with a grid=30 resolution.

```python
def compute_roa(A: np.ndarray, B: np.ndarray,
                P: np.ndarray, Q: np.ndarray, R: np.ndarray,
                N: int, rx: float, ru: float, rf: float,
                grid_dim: int = 5, max_steps: int = 20,
                tol: float = 1e-2):
    """Compute a region of attraction."""

    roa = np.zeros((grid_dim, grid_dim))
    xs = np.linspace(-rx, rx, grid_dim)
    for i, x1 in enumerate(xs):
        for j, x2 in enumerate(xs):
            x = np.array([x1, x2])
            # ################## PART (b): YOUR CODE BELOW ##################

            # INSTRUCTIONS: Simulate the closed-loop system for `max_steps`,
            #               stopping early only if the problem becomes
            #               infeasible or the state has converged close enough
            #               to the origin. If the state converges, flag the
            #               corresponding entry of `roa` with a value of `1`.

            for _ in range(max_steps):
                u_mpc, status = do_mpc(x,A,B,P,Q,R,N,rx,ru,rf)[1:]

                if status == 'infeasible':
                    break
                else:
                    x = A@x + B@u_mpc[0,:]
                    if np.linalg.norm(x) <= tol:
                        roa[i,j] = 1
                        break

            # ######################## END PART (b) ########################
    return roa
```

Figure 21: P4 code (b)

Problem 5: Terminal ingredients

Consider the discrete-time LTI system $x_{k+1} = Ax_k + Bu_k$ with

$$A = \begin{bmatrix} 0.9 & 0.6 \\ 0 & 0.8 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

We want to synthesize a model predictive controller to regulate the system to the origin while minimizing the quadratic cost function

$$J(x_{0:N}, u_{0:N-1}) = x_N^\mathsf{T} P x_N + \sum_{k=0}^{N-1} \Big(x_k^\mathsf{T} Q x_k + u_k^\mathsf{T} R u_k \Big),$$

subject to $\|x_k\|_2 \leq r_x$, $\|u_k\|_2 \leq r_u$, and $x_N \in \mathcal{X}_f$. For this problem, set $N = 4$, $r_x = 5$, $r_u = 1$, $Q = I$ and $R = I$.

Recall from lecture that the design of the terminal ingredients $\mathcal{X}_f$ and P is critical to guaranteeing the asymptotic stability and persistent feasibility of the resulting closed-loop system under receding horizon control.

(a) For this particular problem, explain why and how we can design $\mathcal{X}_f$ and P in an open-loop manner, i.e., by only considering the uncontrolled system $x_{k+1} = Ax_k$.
ANSWER: A is asymptotically stable, therefore, we can use Ax to construct the maximum positive invariant set.

We want to find as large of a positive invariant set $\mathcal{X}_f$ for $x_{k+1} = Ax_k$ as possible that satisfies the state constraints. While maximal positive invariant sets may be computed via iterative methods using tools from computational geometry[4], we restrict our search to ellipsoids of the form

$$\mathcal{X}_f = \{x \in \mathbb{R}^n \mid x^\mathsf{T} P x \leq 1\}$$

for $P \succ 0$. Since $\text{vol}(\mathcal{X}_f) \sim \sqrt{\det(P^{-1})}$, we can formulate our search for the largest ellipsoidal $\mathcal{X}_f$ as the semi-definite program (SDP)

$$\begin{aligned} \underset{P}{\text{maximize}} \quad & \log\det(P^{-1}) \\ \text{subject to} \quad & P \succ 0 \\ & A^\mathsf{T} P A - P \prec 0 \\ & I - r_x^2 P \prec 0 \end{aligned}.$$

[4]See the MPT3 library (https://www.mpt3.org/) for some examples tailored to model predictive control.

(b) Prove that $A^\mathsf{T}PA - P \prec 0$ and $I - r_x^2 P \prec 0$ together are sufficient conditions for $\mathcal{X}_f$ to be a positive invariant set satisfying the state constraints.

ANSWER:

$$A^\mathsf{T}PA - P \prec 0$$

$$x^\mathsf{T}A^\mathsf{T}PAx \prec x^\mathsf{T}Px \leq 1$$

The condition is sufficient to satisfy the positive invariant set X_f.

$$I - r_x^2 P \prec 0$$

$$I \prec r_x^2 P$$

$$x^\mathsf{T}x \prec r_x^2 x^\mathsf{T}Px \leq r_x^2$$

$$||x||_2^2 \leq r_x^2$$

$$||x||_2 \leq r_x$$

The condition satisfies the constraint.

(c) For a maximization problem, we want the objective to be a concave function. Unfortunately, the given SDP is not concave since $\log\det(Q^{-1}) = -\log\det(Q)$ is convex with respect to its argument $Q \succ 0$. Reformulate the given SDP in P as a concave SDP in $M \coloneqq P^{-1}$.

Hint: Use Schur's complement lemma, which states that for any conformable matrices A, B, and C where A is invertible,

$$A \succ 0 \text{ and } C - B^\mathsf{T} A^{-1} B \succ 0 \iff \begin{bmatrix} A & B \\ B^\mathsf{T} & C \end{bmatrix} \succ 0.$$

ANSWER: Swap P^{-1} to M. Since if P is positive definite matrix, it's inverse P^{-1} will still be positive definite, which is M.

$$A^\mathsf{T} M^{-1} A - M^{-1} \prec 0$$

$$M A^\mathsf{T} M^{-1} A M - M \prec 0$$

$$M - M A^\mathsf{T} M^{-1} A M \succ 0$$

From the above equation, we can know that Schur's complement lemma $B = AM$, $A = M$, $C = M$. Therefore, we will get the following equation. (M will be symmetric matrix)

$$\begin{bmatrix} M & AM \\ MA^\mathsf{T} & M \end{bmatrix} \succ 0$$

For the last condition,

$$I - r_x^2 M^{-1} \prec 0$$

$$r_x^2 M^{-1} \succ I$$

$$M^{-1} \succ \frac{I}{r_x^2}$$

If the eigenvalues of M is λ, the eigenvalues of M^{-1} will be $\frac{1}{\lambda}$.

$$M \prec r_x^2 I$$

In the end, we will have the following concave SDP in M.

$$\begin{aligned} \underset{M}{\text{maximize}} \quad & \log\det(M) \\ \text{subject to} \quad & M \succ 0 \\ & \begin{bmatrix} M & AM \\ MA^\mathsf{T} & M \end{bmatrix} \succ 0 \\ & M - r_x^2 I \prec 0 \end{aligned} .$$

(d) Use NumPy and CVXPY to formulate and solve the SDP for M. Plot the ellipsoids $\mathcal{X}_f$, $A\mathcal{X}_f$, and $\mathcal{X} \coloneqq \{x \mid \|x\|_2^2 \leq r_x^2\}$ in the same figure. You should see that $A\mathcal{X}_f \subseteq \mathcal{X}_f \subseteq \mathcal{X}$. Submit your code and plot, and report $P \coloneqq M^{-1}$ with three decimal places for each entry.

Hint: Consult the CVXPY documentation to help you write your code. Specifically, look at the list of functions in CVXPY (`https://www.cvxpy.org/tutorial/functions/index.html`) and the SDP example (`https://www.cvxpy.org/examples/basic/sdp.html`). You can write any definite constraints in the SDP with analogous semi-definite constraints (i.e., treat "$\succ$" as "`>>`" for the purposes of writing your CVXPY code).

For plotting purposes, you can use the following Python function to generate points on the boundary of a two-dimensional ellipsoid.

ANSWER: Problem 5 code is attached at the end. $P = [[0.041, 0.013], [0.013, 0.198]]$

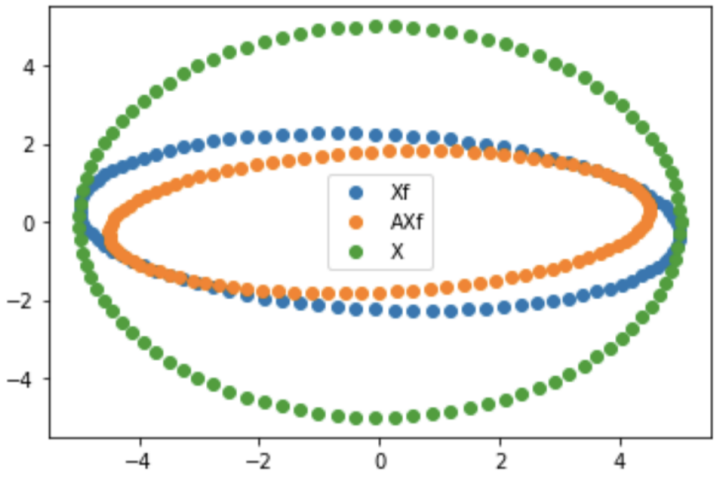


Figure 22: P5 Ellipsoids (d)

(e) Use NumPy and CVXPY to setup the MPC problem, then simulate the system with closed-loop MPC from $x_0 = (0, -4.5)$ for 15 time steps. Overlay the actual state trajectory and the planned trajectories at each time on the plot from part (d). Also, separately plot the actual control trajectory over time in a second plot. Overall, you should have two plots for this entire question. Submit both plots and all of your code.

Hint: Instead of forming the MPC problem in CVXPY during each simulation iteration, form a single CVXPY problem parameterized by the initial state and replace its value before solving (`https://www.cvxpy.org/tutorial/intro/index.html#parameters`).

ANSWER: Problem 5 code is attached at the end.

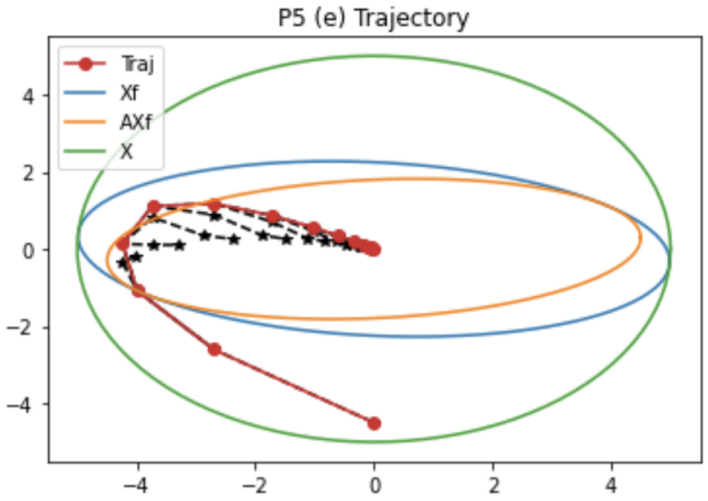


Figure 23: P5 Trajectory (e)

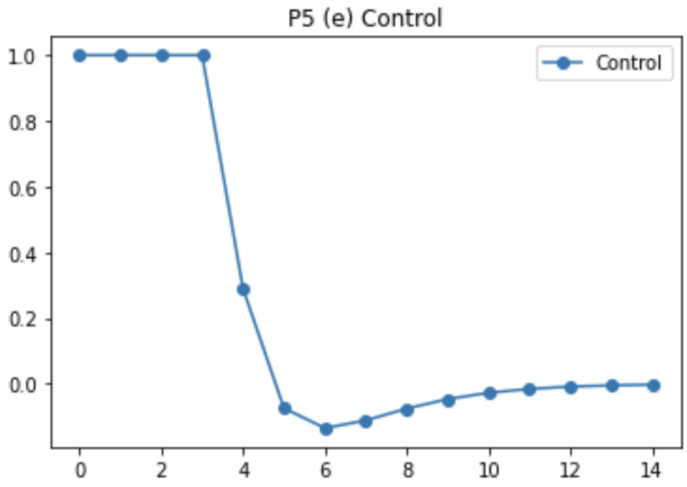


Figure 24: P5 Control (e)

References

[FCTS15] J. F. Fisac, M. Chen, C. J. Tomlin, and S. S. Sastry, *Reach-avoid problems with time-varying dynamics, targets and constraints*, Hybrid Systems: Computation and Control, 2015.

[Mit02] I. M. Mitchell, *Application of level set methods to control and reachability problems in continuous and hybrid systems*, Ph.D. thesis, Stanford University, 2002.

In [48]:
```
!pip install --upgrade cvxpy
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: cvxpy in /usr/local/lib/python3.7/dist-packages (1.2.1)
Requirement already satisfied: osqp>=0.4.1 in /usr/local/lib/python3.7/dist-packages (from cvxpy) (0.6.2.post0)
Requirement already satisfied: ecos>=2 in /usr/local/lib/python3.7/dist-packages (from cvxpy) (2.0.10)
Requirement already satisfied: scs>=1.1.6 in /usr/local/lib/python3.7/dist-packages (from cvxpy) (3.2.0)
Requirement already satisfied: numpy>=1.15 in /usr/local/lib/python3.7/dist-packages (from cvxpy) (1.21.6)
Requirement already satisfied: scipy>=1.1.0 in /usr/local/lib/python3.7/dist-packages (from cvxpy) (1.4.1)
Requirement already satisfied: qdldl in /usr/local/lib/python3.7/dist-packages (from osqp>=0.4.1->cvxpy) (0.1.5.
post2)
```

In [59]:
```
import numpy as np
import cvxpy as cvx
import matplotlib.pyplot as plt
from scipy.linalg import solve_discrete_are
from tqdm.auto import tqdm
from itertools import product
import matplotlib.pyplot as plt
```

In [60]:
```
def generate_ellipsoid_points(M, num_points=100):
    L = np.linalg.cholesky(M)
    θ = np.linspace(0, 2*np.pi, num_points)
    u = np.column_stack([np.cos(θ), np.sin(θ)])
    x = u @ L.T
    return x
```

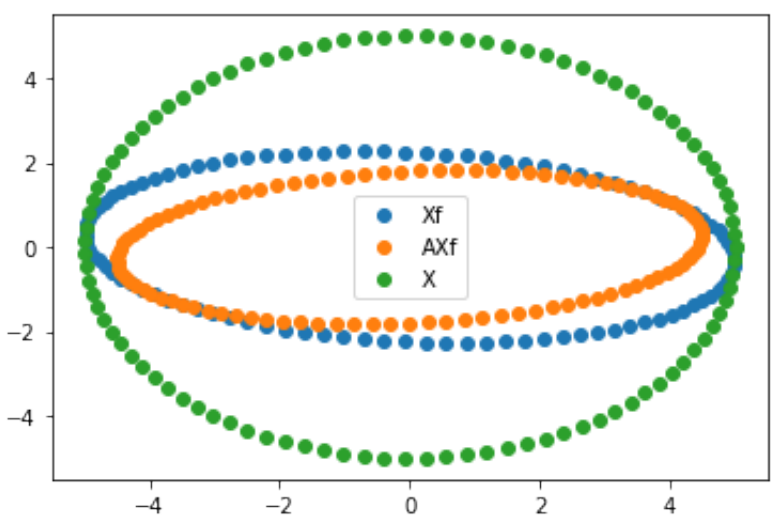
In [61]:
```
def generate_circle_points(num_points=100, rx = 5):
    θ = np.linspace(0, 2*np.pi, num_points)
    x = rx * np.column_stack([np.cos(θ), np.sin(θ)])
    return x
```

In [62]:
```
n,  m = 2, 1
N = 4
A = np.array([[0.9, 0.6],[0, 0.8]])
B = np.array([[0],[1]])

Q = np.eye(n)
R = np.eye(m)

rx = 5
ru = 1
```

In [63]:
```
# P5 (d) Solve the SDP for M

M_cvx = cvx.Variable((n,n), symmetric = True)

objective = cvx.log_det(M_cvx)

constraints = [M_cvx >> 0,
               M_cvx - rx**2 * np.eye(2) << 0,
               cvx.bmat([[M_cvx, A@M_cvx],[M_cvx@A.T, M_cvx]]) >> 0]

prob = cvx.Problem(cvx.Maximize(objective), constraints)
prob.solve()


M = M_cvx.value
status = prob.status
print(M)
P = np.linalg.inv(M)
print(P)
```

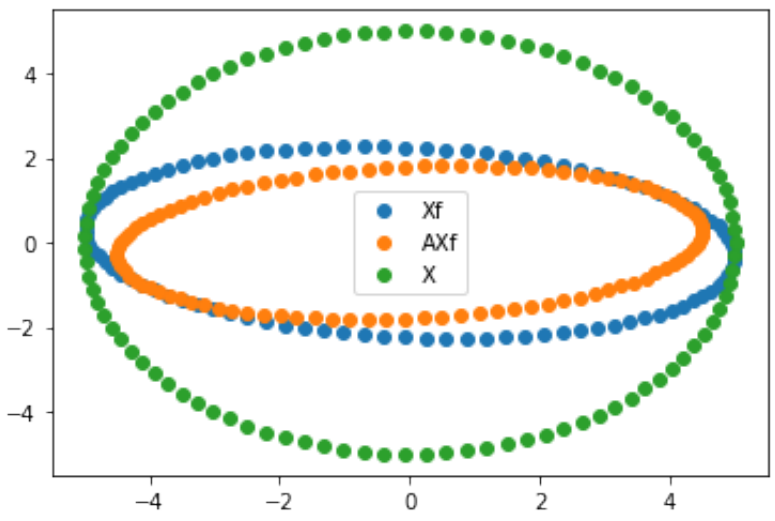
```
[[24.8640275  -1.64239115]
 [-1.64239115  5.15785901]]
[[0.04108286 0.01308181]
 [0.01308181 0.19804447]]
```

In [64]:
```
Xf = generate_ellipsoid_points(M, num_points=100)
AXf = A @ Xf.T
X = generate_circle_points()
plot1 = plt.scatter(Xf[:,0], Xf[:,1])
plot2 = plt.scatter(AXf[0,:], AXf[1,:])
plot3 = plt.scatter(X[:,0], X[:,1])
plt.legend((plot1, plot2, plot3), ('Xf', 'AXf', 'X'))
plt.show()
plt.savefig('P5_d.png', bbox_inches='tight')
```




```
<Figure size 432x288 with 0 Axes>
```

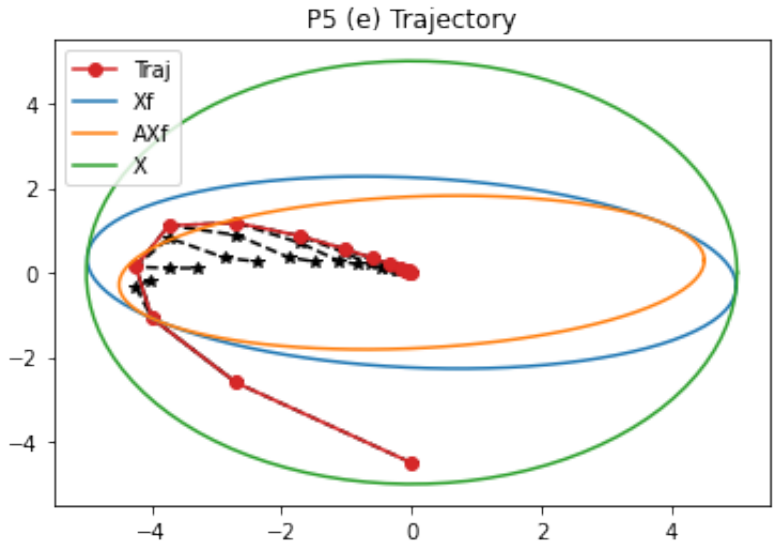
In [65]:
```
AXf = generate_ellipsoid_points(A @ M @ A.T, num_points=100)
plot1 = plt.scatter(Xf[:,0], Xf[:,1])
plot2 = plt.scatter(AXf[:,0], AXf[:,1])
plot3 = plt.scatter(X[:,0], X[:,1])
plt.legend((plot1, plot2, plot3), ('Xf', 'AXf', 'X'))
plt.show()
plt.savefig('P5_d.png', bbox_inches='tight')
```




```
<Figure size 432x288 with 0 Axes>
```

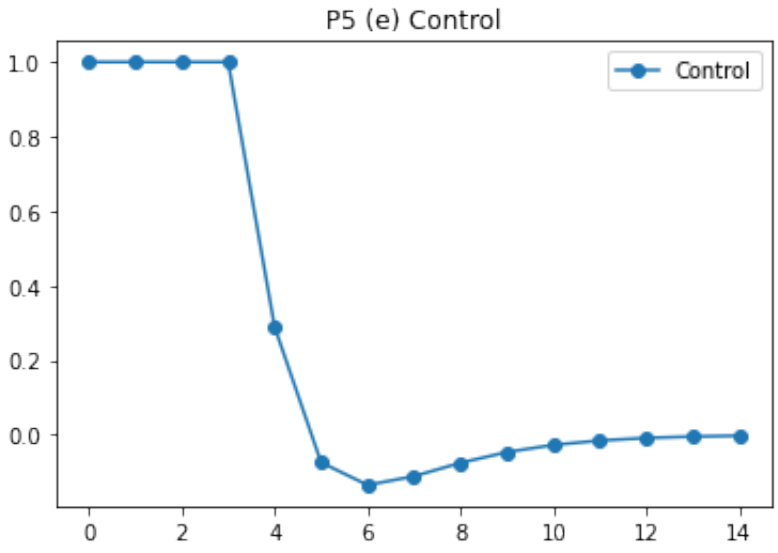
In [66]:
```
def do_mpc(N, x0, A, B, P, Q, R, rx, ru):

  x_cvx = cvx.Variable((N + 1, n))
  u_cvx = cvx.Variable((N, m))

  cost = cvx.quad_form(x_cvx[N],P)

  constraints = [x_cvx[0] == x0]

  for k in range(N):
      cost += cvx.quad_form(x_cvx[k],Q) + cvx.quad_form(u_cvx[k],R)
      constraints += [x_cvx[k+1] == A@x_cvx[k] + B@u_cvx[k]]
      constraints += [cvx.norm(x_cvx[k], 2) <= rx, cvx.norm(u_cvx[k], 2) <= ru]

  constraints += [cvx.norm(x_cvx[N], 2) <= rx]


  prob = cvx.Problem(cvx.Minimize(cost), constraints)
  prob.solve()


  x = x_cvx.value
  u = u_cvx.value
  status = prob.status

  return x, u, status
```

In [67]:
```
T = 15
x0 = np.array([0, -4.5])
x = np.copy(x0)
x_mpc = np.zeros((T, N + 1, n))
u_mpc = np.zeros((T, N, m))
for t in range(T):
  x_mpc[t], u_mpc[t], status = do_mpc(N, x, A, B, P, Q, R, rx, ru)
  if status == 'infeasible':
      x_mpc = x_mpc[:t]
      u_mpc = u_mpc[:t]
      break
  x = A@x + B@u_mpc[t, 0, :]
  plt.plot(x_mpc[t, :, 0], x_mpc[t, :, 1], '--*', color='k')
plt.plot(x_mpc[:, 0, 0], x_mpc[:, 0, 1], '-o', label = 'Traj', color = 'C3')
plt.title('P5 (e) Trajectory')

Xf = generate_ellipsoid_points(M, num_points=100)
AXf = generate_ellipsoid_points(A @ M @ A.T, num_points=100)
X = generate_circle_points()
plt.plot(Xf[:,0], Xf[:,1],'-', label = 'Xf', color = 'C0')
plt.plot(AXf[:,0], AXf[:,1],'-', label = 'AXf',color = 'C1')
plt.plot(X[:,0], X[:,1],'-', label = 'X',color = 'C2')

plt.legend()
plt.show()
plt.savefig('P5_e_Traj.png', bbox_inches='tight')
```



<Figure size 432x288 with 0 Axes>

In [68]:
```
plt.plot(u_mpc[:,0], '-o', label = 'Control', color = 'C0')
plt.title('P5 (e) Control')

plt.legend()
plt.show()

plt.savefig('P5_e_Control.png', bbox_inches='tight')
```



<Figure size 432x288 with 0 Axes>