Stanford **Spring 2024**

AA 203: Optimal and Learning-based Control
Homework #3
Due May 21 by 11:59 pm

Problem 1: To solve a stochastic optimization problem with value iteration by formulating it as an MDP.

Problem 2: Gain familiarity with tools for HJ reachability and develop an understanding of sub-level sets in the context of backward reachability.

Problem 3: Understand the basics of feasibility in MPC.

Problem 4: Introduce algorithmic details of designing terminal ingredients for MPC.

3.1 Markovian drone. In this problem, we will apply techniques for solving a Markov Decision Process (MDP) to guide a flying drone to its destination through a storm. The world is represented as an $n \times n$ grid, i.e., the state space is

$$\mathcal{S} \coloneqq \{(x_1, x_2) \in \mathbb{R}^2 \mid x_1, x_2 \in \{0, 1 \ldots, n-1\}\}.$$

In these coordinates, $(0, 0)$ represents the bottom left corner of the map and $(n-1, n-1)$ represents the top right corner of the map. From any location $x = (x_1, x_2) \in \mathcal{S}$, the drone has four possible directions it can move in, i.e.,

$$\mathcal{A} \coloneqq \{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\}.$$

The corresponding state changes for each action are:

- `up`: $(x_1, x_2) \mapsto (x_1, x_2 + 1)$
- `down`: $(x_1, x_2) \mapsto (x_1, x_2 - 1)$
- `left`: $(x_1, x_2) \mapsto (x_1 - 1, x_2)$
- `right`: $(x_1, x_2) \mapsto (x_1 + 1, x_2)$

Additionally, there is a storm centered at $x_{\text{eye}} \in \mathcal{S}$. The storm's influence is strongest at its center and decays farther from the center according to the equation $\omega(x) = \exp\left(-\frac{\|x - x_{\text{eye}}\|_2^2}{2\sigma^2}\right)$. Given its current state x and action a, the drone's next state is determined as follows:

- With probability $\omega(x)$, the storm will cause the drone to move in a uniformly random direction.
- With probability $1 - \omega(x)$, the drone will move in the direction specified by the action.
- If the resulting movement would cause the drone to leave $\mathcal{S}$, then it will not move at all. For example, if the drone is on the right boundary of the map, then moving right will do nothing.

The quadrotor's objective is to reach $x_{\text{goal}} \in \mathcal{S}$, so the reward function is the indicator function $R(x) = I_{x_{\text{goal}}}(x)$. In other words, the drone will receive a reward of 1 if it reaches the $x_{\text{goal}} \in \mathcal{S}$, and a reward of 0 otherwise. The reward of a trajectory in this infinite horizon problem is a discounted sum of the rewards earned in each timestep, with discount factor $\gamma \in (0, 1)$.

(a) Given $n = 20$, $\sigma = 10$, $\gamma = 0.95$, $x_{\text{eye}} = (15, 15)$, and $x_{\text{goal}} = (19, 9)$, write code that uses value iteration to find the optimal value function for the drone to navigate the storm. Recall that value iteration repeats the Bellman update

$$V(x) \leftarrow \max_{a \in \mathcal{A}} \left(\sum_{x' \in \mathcal{S}} p(x'; x, a)(R(x') + \gamma V(x')) \right)$$

until convergence, where $p(x'; x, a)$ is the probability distribution of the next state being x' after taking action a in state x, and R is the reward function. Plot a heatmap of the optimal value function obtained by value iteration over the grid $\mathcal{S}$, with $x = (0, 0)$ in the bottom left corner, $x = (n - 1, n - 1)$ in the top right corner, the x_1-axis along the bottom edge, and the x_2-axis along the left edge.

(b) Recall that a policy π is a mapping $\pi : \mathcal{S} \to \mathcal{A}$ where $\pi(x)$ specifies the action to be taken should the drone find itself in state x. An optimal value function V^* induces an optimal policy π^* such that

$$\pi^*(x) \in \arg\max_{a \in \mathcal{A}} \left(\sum_{x' \in \mathcal{S}} p(x'; x, a)(R(x') + \gamma V^*(x')) \right)$$

Use the value function you computed in part (a) to compute an optimal policy. Then, use this policy to simulate the MDP for $N = 100$ time steps with the state initialized at $x = (0, 19)$. Plot the policy as a heatmap where the actions $\{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\}$ correspond to the values $\{0, 1, 2, 3\}$, respectively. Plot the simulated drone trajectory overlaid on the policy heatmap, and briefly describe in words what the policy is doing.

3.2 Reach-avoid flight. 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__`).

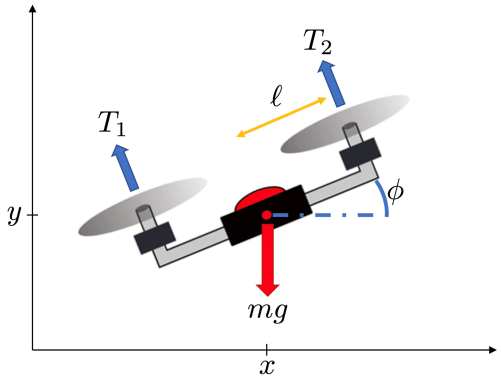


Figure 1: A planar quadrotor.

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

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

[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.

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

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

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

[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).

(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.

(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`.

(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 2, which you may also use to check your work) has the shape that it does.

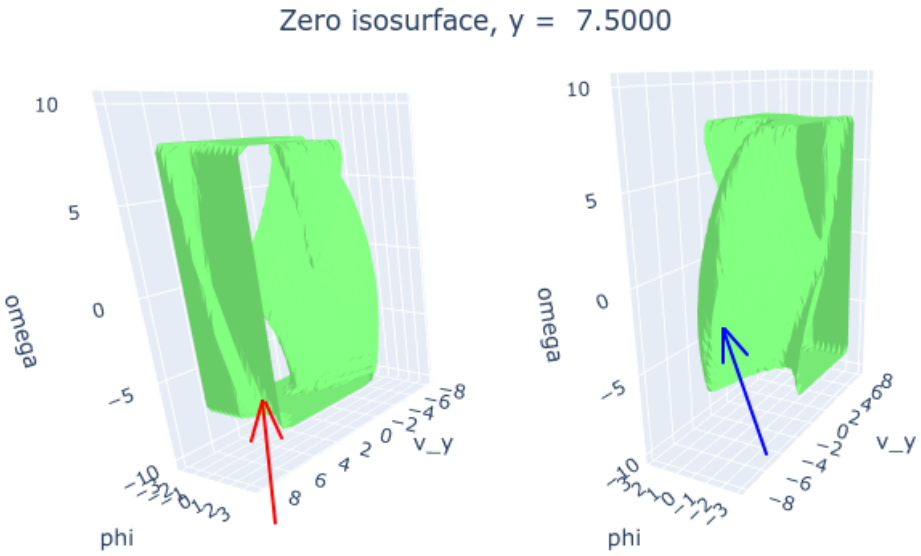


Figure 2: Example zero isosurface views. Can you explain why the red valley (outside the isosurface, i.e., unrecoverable initial conditions) or blue ridge (inside the isosurface, i.e., initial conditions that can reach the target collision-free) exhibit the "tilt" they have by considering the corresponding 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

[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 $h(\mathbf{x})$.

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).

(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.

3.3 MPC feasibility. Consider the discrete-time LTI system

$$x_{t+1} = Ax_t + Bu_t.$$

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

$$J(x,u) = x_T^\mathsf{T} P x_T + \sum_{t=0}^{T-1} \Big(x_t^\mathsf{T} Q x_t + u_t^\mathsf{T} R u_t \Big),$$

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

For this problem, you will work with the starter code in `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 $r_T \geq r_x$ 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.

(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_T. 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. However, you must submit your plot of the ROAs with at least `grid_dim = 30`.

3.4 Terminal ingredients. Consider the discrete-time LTI system $x_{t+1} = Ax_t + Bu_t$ 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

$$J(x, u) = x_T^\mathsf{T} P x_T + \sum_{t=0}^{T-1} \Big(x_t^\mathsf{T} Q x_t + u_t^\mathsf{T} R u_t \Big),$$

with $Q \succ 0$, $R \succ 0$, and $P \succ 0$, subject to $\|x_t\|_2 \leq r_x$, $\|u_t\|_2 \leq r_u$, and $x_T \in \mathcal{X}_T$. For this problem, set $N = 4$, $r_x = 5$, $r_u = 1$, $Q = I$ and $R = I$.

Recall from lecture that the terminal ingredients $\mathcal{X}_T$ and P are critical to recursive feasibility and stability of the resulting closed-loop system under receding horizon control.

(a) For this particular problem, explain why and how we can design $\mathcal{X}_T$ and P in an open-loop manner, i.e., by only considering the uncontrolled system $x_{t+1} = Ax_t$. You only need to describe what properties of $\mathcal{X}_T$ and P your method must ensure to guarantee recursive feasibility and stability of the resulting closed-loop system with MPC feedback.

For the remainder of this problem, set $P = I$ for simplicity.

We want to find as large of a positive invariant set $\mathcal{X}_T$ for $x_{t+1} = Ax_t$ 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

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

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

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

Critically, each constraint in a convex or concave SDP is a *linear matrix inequality (LMI)*.

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

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

Hint: You should use

- the fact that $B \preceq C$ if and only if $ABA \preceq ACA$ for symmetric A, B, and C where $A \succ 0$,
- the fact that $A \preceq \gamma I$ if and only if $A^{-1} \succeq \frac{1}{\gamma} I$ for $A \succ 0$ and $\gamma > 0$, and

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

- Schur's complement lemma, which states that

$$C - B^\mathsf{T} A^{-1} B \succeq 0 \iff \begin{bmatrix} A & B \\ B^\mathsf{T} & C \end{bmatrix} \succeq 0$$

  for any conformable matrices A, B, and C where $A \succ 0$.

(d) Use NumPy and CVXPY to formulate and solve the SDP for M. Plot the ellipsoids $\mathcal{X}_T$, $A\mathcal{X}_T$, and $\mathcal{X} := \{x \mid \|x\|_2^2 \leq r_x^2\}$ in the same figure. You should see that $A\mathcal{X}_T \subseteq \mathcal{X}_T \subseteq \mathcal{X}$. Submit your code and plot, and report $W := M^{-1}$ 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.

```python
import numpy as np

def generate_ellipsoid_points(M, num_points=100):
    """Generate points on a 2-D ellipsoid.

    The ellipsoid is described by the equation
        `{ x | x.T @ inv(M) @ x <= 1 }`,
    where `inv(M)` denotes the inverse of the matrix argument `M`.

    The returned array has shape (num_points, 2).
    """
    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
```

(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).

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.

Problem 3.1: Markovian Drone

a) Use an MDP to guide a flying drone to its destination through a storm

$n \times n$ world grid with state space $S = \{(x_1, x_2) \in \mathbb{N}^2 \mid x_1, x_2 \in \{0, 1, \dots n-1\}\}$

From any location $x = (x_1, x_2) \in S$, drone can move in $A = \{\text{up, down, left, right}\}$

Storm centered at $x_{eye} \in S$ ← influence is strongest at center and decays as $w(x) = \exp\left(-\frac{\|x - x_{eye}\|_2^2}{2\sigma^2}\right)$

Given current state x and action a, next state is determined as

- Probability $w(x)$ → storm causes drone to move in a uniformly random direction
- Probability $1 - w(x)$ → drone moves in direction of a
- If movement causes the drone to leave S, then the drone will not move at all

Goal for drone is to reach $x_{goal} \in S$, reward is the indicator function $R(x) = \mathbb{I}_{x_{goal}}(x)$

b) Reward of a trajectory is a discounted sum of the rewards earned in each timestep, with discount factor $\gamma \in (0, 1)$

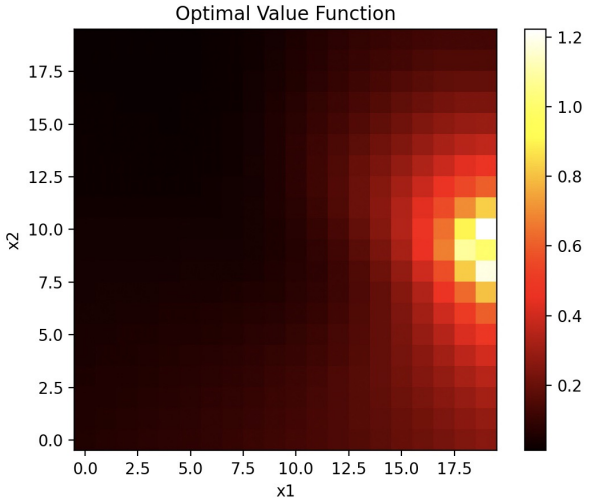
$n = 20$, $\sigma = 10$, $\gamma = 0.95$, $x_{eye} = (15, 15)$, $x_{goal} = (19, 0)$ → use value iteration to find the optimal value function for the drone to navigate the storm

Bellman update: $V(x) \leftarrow \max_{a \in A} \left(\sum_{x' \in S} p(x'; x, a) \left(R(x') + \gamma V(x') \right) \right)$

$p(x'; x, a)$ is the probability distribution of the next state being x' after taking action a in state x, and R is the reward function

See attached code for implementation

Heatmap of the optimal value function from value iteration



b) Policy π is a mapping $\pi: S \to A$, where $\pi(x)$ specifies the action to be taken should the drone find itself in state x

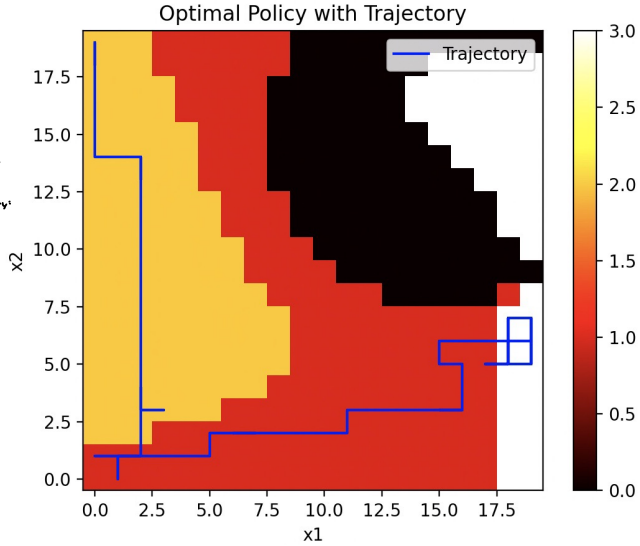
An optimal value function V^* induces an optimal policy π^* such that

$$\pi^*(x) \in \operatorname*{argmax}_{a \in A} \left(\sum_{x' \in S} p(x'; x, a) \left(R(x') + \gamma V^*(x') \right) \right)$$

Find optimal policy based on previous optimal value function

See attached code for implementation

Heatmap of optimal policy with trajectory:



The policy is staying as far away from the center of the storm as possible, to increase the chance that the desired action is actually taken by the drone since the probability of being deterred by the storm increases the closer you are to the center of the storm. Since the eye of the storm is at $x_{eye} = (15, 15)$, the optimal policy chooses to move the drone downwards first and then to the right to avoid the storm.

```python
import numpy as np
import matplotlib.pyplot as plt

def get_storm_prob(x, x_eye, sigma):
    return np.exp(-np.linalg.norm(np.array(x) - np.array(x_eye))**2 / (2 * sigma**2))

# indicator function
def get_reward(x, x_goal):
    if x == x_goal:
        return 1
    return 0

def get_action_probs(x, x_eye, sigma, action):
    w = get_storm_prob(x, x_eye, sigma)
    action_probs = {"up": 0.0, "down": 0.0, "left": 0.0, "right": 0.0}
    for a in action_probs.keys():
        if a == action:
            action_probs[a] = 1 - w
        else:
            action_probs[a] = w / 3
    return action_probs

def get_next_state(x, action, n):
    i, j = x
    if action == "up":
        return (max(i-1, 0), j)
    if action == "down":
        return (min(i+1, n-1), j)
    if action == "left":
        return (i, max(j-1, 0))
    if action == "right":
        return (i, min(j+1, n-1))

def bellman_update(V, x, x_eye, x_goal, sigma, gamma, n):
    if x == x_goal: # terminal state
        return 1
    max_val = -np.inf
    for action in ["up", "down", "left", "right"]:
        action_probs = get_action_probs(x, x_eye, sigma, action)
        val = 0
        for a, p in action_probs.items():
            x_next = get_next_state(x, a, n)
            val += p * (get_reward(x_next, x_goal) + gamma * V[x_next])
        max_val = max(max_val, val)
    return max_val

def run_value_iteration(V_0, x_eye, x_goal, sigma, gamma, n, eps = 1e-6):
    V = V_0.copy()
    while True:
        V_new = V.copy()
        for i in range(n):
            for j in range(n):
                x = (i, j)
                V_new[x] = bellman_update(V, x, x_eye, x_goal, sigma, gamma, n)
        if np.linalg.norm(V_new - V) < eps:
            break
        V = V_new
    return V

def get_optimal_policy(V_star, x_eye, x_goal, sigma, gamma, n):
    policy = np.zeros((n, n), dtype=f'<U10')
    for i in range(n):
        for j in range(n):
            x = (i, j)
            max_val = -np.inf
            for action in ["up", "down", "left", "right"]:
                action_probs = get_action_probs(x, x_eye, sigma, action)
```

```
                val = 0
                for a, p in action_probs.items():
                    x_next = get_next_state(x, a, n)
                    val += p * (get_reward(x_next, x_goal) + gamma * V_star[x_next])
                if val > max_val:
                    max_val = val

                    policy[x[0], x[1]] = action
    return policy

def simulate_trajectory(x_0, pi_star, x_eye, x_goal, sigma, gamma, n, N):
    x = x_0
    trajectory = [x]
    for _ in range(N): # run the MDP for N steps
        action = pi_star[x] # ideal action
        action_probs = get_action_probs(x, x_eye, sigma, action)
        action = np.random.choice(["up", "down", "left", "right"],
p=list(action_probs.values())) # get the next action based on the storm probabilities
        x = get_next_state(x, action, n) # move to the next state
        trajectory.append(x)
    return trajectory

def convert_policy_to_numeric(pi_star, n=20):
    pi_star_numeric = np.zeros((n, n), dtype=int)
    for i in range(n):
        for j in range(n):
            if pi_star[i, j] == "up":
                pi_star_numeric[i, j] = 0
            elif pi_star[i, j] == "down":
                pi_star_numeric[i, j] = 1
            elif pi_star[i, j] == "left":
                pi_star_numeric[i, j] = 2
            elif pi_star[i, j] == "right":
                pi_star_numeric[i, j] = 3
    return pi_star_numeric

def main():
    # Constants
    n = 20
    sigma = 10
    gamma = 0.95
    x_eye = (15, 15)
    x_goal = (19, 9)

    V_0 = np.zeros((n, n))

    V_star = run_value_iteration(V_0, x_eye, x_goal, sigma, gamma, n)

    # Plot heatmap, part a
    plt.imshow(V_star.T, cmap='hot', origin='lower')
    plt.colorbar()
    plt.title('Optimal Value Function')
    plt.xlabel('x1')
    plt.ylabel('x2')

    pi_star = get_optimal_policy(V_star, x_eye, x_goal, sigma, gamma, n)
    pi_star_numeric = convert_policy_to_numeric(pi_star, n)

    N = 100
    x_init = (0, 19)
    trajectory = simulate_trajectory(x_init, pi_star, x_eye, x_goal, sigma, gamma, n, N)

    # Plot trajectory, part b
    plt.figure()
    plt.imshow(pi_star_numeric.T, cmap='hot', origin='lower')
    plt.plot([x[0] for x in trajectory], [x[1] for x in trajectory], 'b-')
    plt.colorbar()
```

```
    plt.title('Optimal Policy with Trajectory')
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.legend(['Trajectory'])
    plt.show()


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

Problem 3.3: Reach-avoid Flight

a) Self-righting drone 6-D dynamics

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

State is the vertical plane (x, y), translational velocity (v_x, v_y), pitch ϕ, pitch rate ω

Controls are thrusts T_1, T_2

Approach in an HJB, CT OP formulation

Reduced 4D dynamics: $x := (v_x, v_y, \phi, \omega) \in \mathbb{R}^4$

Target set: $\mathcal{T} = [5, 7] \times [-1, 1] \times [-\frac{\pi}{12}, \frac{\pi}{12}] \times [-1, 1] \subset \mathbb{R}^4$ ← once the controller reaches this set, another controller can take over to maintain level flight

Quadrotor must stay within operational envelope: $\mathcal{E} = [-1, 9] \times [-6, 6] \times [-\infty, \infty] \times [-8, 8]$

Reach-avoid problem: reaching the target set $\mathcal{T}$ while avoiding the obstacle set $\mathcal{E}^c$

Lipschitz continuous fns $h(x), e(x)$ s.t. $x \in \mathcal{T} \Rightarrow h(x) \leq 0$, $x \in \mathcal{E} \Rightarrow e(x) \leq 0$

i.e. $\mathcal{T}, \mathcal{E}$ are the zero-sublevel sets of h, e respectively, then it is shown that the value function $V(\cdot, t)$

$$V(x_0, t_0) = \min_{u(\cdot)} \min_{T \in [t_0, 0]} h(x(T))$$

$$\text{s.t. } \dot{x}(T) = f(x(T), u(T)) \quad \forall T \in [t_0, 0]$$

$$x(T) \in \mathcal{E} \quad \forall T \in [t_0, 0]$$

$$x(t_0) = x_0$$

satisfies the HJB PDE:

$$\max\left\{ \frac{\partial V}{\partial t}(x,t) + \min\{0, H(x, \nabla_x V(x,t))\}, e(x) - V(x,t) \right\} = 0$$

where $H(x,p) = \min_u p^T f(x,u)$,

$$V(x,0) = \max\{h(x), e(x)\}$$

Derive locally optimal action that minimizes the Hamiltonian subject to constraints: $T_1, T_2 \in [0, T_{max}]$

for arbitrary x, p, compute

$$u^* = \arg\min_u p^T f(x,u)$$

Hamiltonian: $H(x,p) = \min_u p^T f(x,u)$, $p = \nabla_x V$ is the gradient of the value function wrt the state variables

$p = (p_1, p_2, p_3, p_4)$

$$\therefore p^T f(x,u) = p_1 \dot{v}_x + p_2 \dot{v}_y + p_3 \dot{\phi} + p_4 \dot{\omega}$$

$$= p_1 v_x + p_2 \left(\frac{(T_1+T_2)\cos\phi - C_D v_y}{m} - g \right) + p_3 \omega + p_4 \left(\frac{(T_2-T_1)\ell - C_D \omega}{I_{yy}} \right)$$

∴ To find the optimal u, we need to optimize wrt T_1 and T_2 → find $\frac{\partial}{\partial T_1}, \frac{\partial}{\partial T_2} = 0$

$$\therefore u^* = \arg\min_{T_1, T_2} \left(p_1 v_x + p_2 \left(\frac{(T_1+T_2)\cos\phi - C_D v_y}{m} - g \right) + p_3 \omega + p_4 \left(\frac{(T_2-T_1)\ell - C_D \omega}{I_{yy}} \right) \right)$$

The Hamiltonian takes the form

$$u^* = \arg\min_{T_1, T_2} \left(p_2 \frac{(T_1+T_2)\cos\phi}{m} + p_4 \frac{(T_2-T_1)\ell}{I_{yy}} + (\text{constant terms wrt } T_1, T_2) \right)$$

$$= \arg\min_{T_1, T_2} \left(\frac{p_2 T_1 \cos\phi}{m} + \frac{p_2 T_2 \cos\phi}{m} + \frac{p_4 T_2 \ell}{I_{yy}} - \frac{p_4 T_1 \ell}{I_{yy}} + (\text{constant terms wrt } T_1, T_2) \right)$$

$$= \arg\min_{T_1, T_2} \left(T_1 \left(\frac{p_2 \cos\phi}{m} - \frac{p_4 \ell}{I_{yy}} \right) + T_2 \left(\frac{p_2 \cos\phi}{m} + \frac{p_4 \ell}{I_{yy}} \right) + (\text{constant terms wrt } T_1, T_2) \right)$$

Hence, from this, we have 2 cases each for setting T_1 and T_2 to determine u^*

T_1: If $\frac{p_2 \cos\phi}{m} - \frac{p_4 \ell}{I_{yy}} \geq 0$, $T_1 = 0$

If $\frac{p_2 \cos\phi}{m} - \frac{p_4 \ell}{I_{yy}} < 0$, $T_1 = T_{max}$

T_2: If $\frac{p_2 \cos\phi}{m} + \frac{p_4 \ell}{I_{yy}} \geq 0$, $T_2 = 0$

If $\frac{p_2 \cos\phi}{m} + \frac{p_4 \ell}{I_{yy}} < 0$, $T_2 = T_{max}$

See attached code for implementation of optimal control

b) Functional form for $h(x)$ such that $x \in T \iff h(x) \leq 0$

$a(x) \leq 0 \ \wedge \ b(x) \leq 0 \iff \max\{a(x), b(x)\} \leq 0$ ← if you have multiple constraints represented as zero-sublevel sets of multiple functions, then the conjunction of the constraints may be a pointwise maximum of the fns

Our target set is defined as

$$T = \underbrace{[3,7]}_{y} \times \underbrace{[-1,1]}_{v_y} \times \underbrace{\left[-\tfrac{\pi}{12}, \tfrac{\pi}{12}\right]}_{\phi} \times \underbrace{[-1,1]}_{\omega} \subset \mathbb{R}^4$$

$\therefore \ y \in [3,7], \quad v_y \in [-1,1], \quad \phi \in \left[-\tfrac{\pi}{12}, \tfrac{\pi}{12}\right], \quad \omega \in [-1,1]$

↳ $\max(3-y, y-7) \leq 0$

↳ $\max(-1-v_y, v_y-1) \leq 0$

↳ $\max\left(-\tfrac{\pi}{12}-\phi, \phi-\tfrac{\pi}{12}\right) \leq 0$

↳ $\max(-1-\omega, \omega-1) \leq 0$

By the hint, we can take the pointwise maximum of all the target set points (maximum of the original fns at that point)

$\therefore$ $h(x) = \max\left(\max(3-y, y-7), \ \max(-1-v_y, v_y-1), \ \max\left(-\tfrac{\pi}{12}-\phi, \phi-\tfrac{\pi}{12}\right), \ \max(-1-\omega, \omega-1)\right)$

See code attached for implementation of target_set

c) Functional form for $e(x)$ such that $x \in \mathcal{E}, \quad e(x) \leq 0$

$$\mathcal{E} = \underbrace{[1,9]}_{y} \times \underbrace{[-6,6]}_{v_y} \times \underbrace{[-\infty,\infty]}_{\phi} \times \underbrace{[-8,8]}_{\omega}$$

$y \in [1,9], \quad v_y \in [-6,6], \quad \phi \in [-\infty,\infty], \quad \omega \in [-8,8]$

$\max(1-y, y-9) \leq 0$

$\max(-6-v_y, v_y-6) \leq 0$

$\max(-\infty-\phi, \phi-\infty) \leq 0$

$\max(-8-\omega, \omega-8) \leq 0$

$\therefore$ $e(x) = \max(\max(1-y, y-9), \ \max(-6-v_y, v_y-6), \ \max(-\infty-\phi, \phi-\infty), \ \max(-8-\omega, \omega-8))$

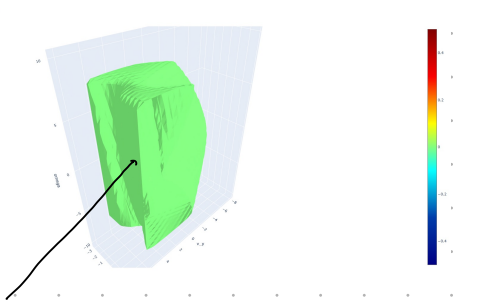
↳ ϕ basically has no constraint,

$e(x) = \max(\max(1-y, y-9), \ \max(-6-v_y, v_y-6), \ 0, \ \max(-8-\omega, \omega-8))$

See attached code for envelope_set implementation

d) Run rest of script to compute $V(x,-5)$

Isosurface Plot:



When examining this planar area on the isosurface, which is outside the isosurface, this area represents initial conditions from which the quadrotor can't reach the target set without either crashing or violating a constraint. This area is described by states where the quadrotor has a high angular velocity and very extreme pitch, which makes it difficult for the quadrotor to recover because of control constraints and time constraints.

e) Some pros of using DP are that you can come to a globally optimal solution from a wide set of initial conditions. This policy can be pre-computed and stored for real-time execution. In addition, DP can effectively handle the potentially complex dynamics of the quadrotor. Some cons of a DP approach are that computing an optimal policy often takes a lot of computation, the policies are not very adaptable to new environments and obstacles, and can result in extreme control strategies that aren't smooth. In comparison to MPC, MPC is more flexible and can adapt to dynamic environments, provide smoother control inputs, and can handle constraints added in real-time. While MPC has these advantages, it suffers from requiring to solve an optimization problem at each timestep, which is computationally expensive and may not result in a globally optimal solution.

```
"""
Starter code for the problem "Hamilton-Jacobi reachability".

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

import os

import jax
import jax.numpy as jnp
import numpy as np

import hj_reachability as hj

import matplotlib.pyplot as plt
from animations import animate_planar_quad

# Define problem ingredients (exercise parts (a), (b), (c)).


class PlanarQuadrotor:

    def __init__(self):
        # Dynamics constants
        # yapf: disable
        self.g = 9.807          # gravity (m / s**2)
        self.m = 2.5            # mass (kg)
        self.l = 1.0            # half-length (m)
        self.Iyy = 1.0          # moment of inertia about the out-of-plane axis (kg * m**2)
        self.Cd_v = 0.25        # translational drag coefficient
        self.Cd_phi = 0.02255   # rotational drag coefficient
        # yapf: enable

        # Control constraints
        self.max_thrust_per_prop = (
            0.75 * self.m * self.g
        )  # total thrust-to-weight ratio = 1.5
        self.min_thrust_per_prop = (
            0  # at least until variable-pitch quadrotors become mainstream :D
        )

    def full_dynamics(self, full_state, control):
        """Continuous-time dynamics of a planar quadrotor expressed as an ODE."""
        x, v_x, y, v_y, phi, omega = full_state
        T_1, T_2 = control
        return jnp.array(
            [
                v_x,
                (-(T_1 + T_2) * jnp.sin(phi) - self.Cd_v * v_x) / self.m,
                v_y,
                ((T_1 + T_2) * jnp.cos(phi) - self.Cd_v * v_y) / self.m - self.g,
                omega,
                ((T_2 - T_1) * self.l - self.Cd_phi * omega) / self.Iyy,
            ]
        )

    def dynamics(self, state, control):
        """Reduced (for the purpose of reachable set computation) continuous-time dynamics of a
planar quadrotor."""
        y, v_y, phi, omega = state
        T_1, T_2 = control
        return jnp.array(
            [
                v_y,
                ((T_1 + T_2) * jnp.cos(phi) - self.Cd_v * v_y) / self.m - self.g,
                omega,
                ((T_2 - T_1) * self.l - self.Cd_phi * omega) / self.Iyy,
```

```
            ]
        )

    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
        _, vy, phi, omega = state
        p1, p2, p3, p4 = grad_value

        min_H = jnp.inf
        optimal_control = jnp.array([self.min_thrust_per_prop, self.min_thrust_per_prop])
        possible_vals = [(self.min_thrust_per_prop, self.min_thrust_per_prop),
(self.min_thrust_per_prop, self.max_thrust_per_prop), (self.max_thrust_per_prop,
self.min_thrust_per_prop), (self.max_thrust_per_prop, self.max_thrust_per_prop)]
        for T1, T2 in possible_vals:
            dyn = self.dynamics(state, [T1, T2])
            H = grad_value.T @ dyn
            #jax.debug.print("Value inside JIT: {}", H)
            optimal_control = jnp.where(H < min_H, jnp.array([T1, T2]), optimal_control) #
flip these
            min_H = jnp.where(H < min_H, H, min_H)

        return optimal_control

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

    def hamiltonian(self, state, time, value, grad_value):
        """Evaluates the HJ PDE Hamiltonian."""
        del time, value  # unused
        control = self.optimal_control(state, grad_value)
        return grad_value @ self.dynamics(state, control)

    def partial_max_magnitudes(self, state, time, value, grad_value_box):
        """Computes the max magnitudes of the Hamiltonian partials over the `grad_value_box` in
each dimension."""
        del time, value, grad_value_box  # unused
        y, v_y, phi, omega = state
        return jnp.array(
            [
                jnp.abs(v_y),
                (
                    2 * self.max_thrust_per_prop * jnp.abs(jnp.cos(phi))
                    + self.Cd_v * jnp.abs(v_y)
                )
                / self.m
                + self.g,
                jnp.abs(omega),
                (
                    (self.max_thrust_per_prop - self.min_thrust_per_prop) * self.l
                    + self.Cd_phi * jnp.abs(omega)
                )
                / self.Iyy,
            ]
```

```
        )


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, v_y, phi, omega = state
    y_h = jnp.maximum(3 - y, y - 7)
    v_y_h = jnp.maximum(-1 - v_y, v_y - 1)
    phi_h = jnp.maximum(-(np.pi/12) - phi, phi - (np.pi/12))
    omega_h = jnp.maximum(-1 - omega, omega - 1)
    h_x = jnp.maximum(jnp.maximum(y_h, v_y_h), jnp.maximum(phi_h, omega_h))
    return h_x
    ##################################################################################


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, v_y, phi, omega = state
    y_e = jnp.maximum(1 - y, y - 9)
    v_y_e = jnp.maximum(-6 - v_y, v_y - 6)
    phi_e = 0
    omega_e = jnp.maximum(-8 - omega, omega - 8)
    e_x = jnp.maximum(jnp.maximum(y_e, v_y_e), jnp.maximum(phi_e, omega_e))
    return e_x
    ##################################################################################


def test_optimal_control(n=10, seed=0):
    planar_quadrotor = PlanarQuadrotor()
    optimal_control = jax.jit(planar_quadrotor.optimal_control)
    np.random.seed(seed)
    states = 5 * np.random.normal(size=(n, 4))
    grad_values = np.random.normal(size=(n, 4))
    try:
        for state, grad_value in zip(states, grad_values):
            if not jnp.issubdtype(
                optimal_control(state, grad_value).dtype, jnp.floating
            ):
                raise ValueError(
                    "`PlanarQuadrotor.optimal_control` must return a `float` array (i.e., not
`int`)."
                )
            opt_hamiltonian_value = grad_value @ planar_quadrotor.dynamics(
                state, optimal_control(state, grad_value)
            )
            for T_1 in (
                planar_quadrotor.min_thrust_per_prop,
                planar_quadrotor.max_thrust_per_prop,
            ):
```

```
                for T_2 in (
                    planar_quadrotor.min_thrust_per_prop,
                    planar_quadrotor.max_thrust_per_prop,
                ):
                    hamiltonian_value = grad_value @ planar_quadrotor.dynamics(
                        state, np.array([T_1, T_2])
                    )
                    if opt_hamiltonian_value > hamiltonian_value + 1e-4:
                        raise ValueError(
                            "Check your logic for `PlanarQuadrotor.optimal_control`; with "
                            f"`state` {state} and `grad_value` {grad_value}, got optimal
control"
                            f"{optimal_control(state, grad_value)} with corresponding
Hamiltonian value "
                            f"{opt_hamiltonian_value:7.4f} but {np.array([T_1, T_2])} has a
lower corresponding "
                            f"value {hamiltonian_value:7.4f}."
                        )
    except (jax.errors.JAXTypeError, jax.errors.JAXIndexError, AssertionError) as e:
        print(
            "`PlanarQuadrotor.optimal_control` must be implemented using only `jnp`
operations; "
            "`np` may only be used for constants, "
            "and `jnp.where` must be used instead of native python control flow
(`if`/`else`)."
        )
        raise e


def test_target_set():
    try:
        in_states = [
            np.array([5.0, 0.0, 0.0, 0.0]),
            np.array([6.0, 0.1, 0.1, 0.1]),
            # feel free to add test cases
        ]
        out_states = [
            np.array([2.0, 0.0, 0.0, 0.0]),
            np.array([5.0, 2.0, 0.0, 0.0]),
            np.array([5.0, 0.0, 2.0, 0.0]),
            np.array([5.0, 0.0, 0.0, 2.0]),
            # feel free to add test cases
        ]
        for x in in_states:
            if not jnp.issubdtype(target_set(x).dtype, jnp.floating):
                raise ValueError(
                    "`target_set` must return a `float` scalar (i.e., not `int`)."
                )
            if target_set(x) > 0:
                raise ValueError(
                    f"Check your logic for `target_set`; for `state` {x} (in) you have
target_set(state) = "
                    f"{target_set(x)}."
                )
        for x in out_states:
            if target_set(x) <= 0:
                raise ValueError(
                    f"Check your logic for `target_set`; for `state` {x} (out) you have
target_set(state) = "
                    f"{target_set(x)}."
                )
    except (jax.errors.JAXTypeError, jax.errors.JAXIndexError, AssertionError) as e:
        print(
            "`target_set` must be implemented using only `jnp` operations, "
            "`np` may only be used for constants, "
            "and `jnp.where` must be used instead of native python control flow
(`if`/`else`)."
```

```
        )
        raise e


def test_envelope_set():
    try:
        in_states = [
            np.array([5.0, 0.0, 0.0, 0.0]),
            np.array([7.0, 5.0, 100.0, 6.0]),
            # feel free to add test cases
        ]
        out_states = [
            np.array([0.0, 0.0, 0.0, 0.0]),
            np.array([5.0, 8.0, 0.0, 0.0]),
            np.array([5.0, 0.0, 0.0, 10.0]),
            # feel free to add test cases
        ]
        for x in in_states:
            if not jnp.issubdtype(envelope_set(x).dtype, jnp.floating):
                raise ValueError(
                    "`envelope_set` must return a `float` scalar (i.e., not `int`)."
                )
            if envelope_set(x) > 0:
                raise ValueError(
                    f"Check your logic for `envelope_set`; for `state` {x} (in) you have
envelope_set(state) = "
                    f"{envelope_set(x)}."
                )
        for x in out_states:
            if envelope_set(x) <= 0:
                raise ValueError(
                    f"Check your logic for `envelope_set`; for `state` {x} (out) you have
envelope_set(state) = "
                    f"{envelope_set(x)}."
                )
    except (jax.errors.JAXTypeError, jax.errors.JAXIndexError, AssertionError) as e:
        print(
            "`envelope_set` must be implemented using only `jnp` operations, "
            "`np` may only be used for constants, "
            "and `jnp.where` must be used instead of native python control flow
(`if`/`else`)."
        )
        raise e


test_optimal_control()
print("test_optimal_control complete")
test_target_set()
print("test_target_set complete")
test_envelope_set()
print("test_envelope_set complete")

# Set up problem for use with PDE solver.
planar_quadrotor = PlanarQuadrotor()
state_domain = hj.sets.Box(
    lo=np.array([0.0, -8.0, -np.pi, -10.0]), hi=np.array([10.0, 8.0, np.pi, 10.0])
)
grid_resolution = (
    25,
    25,
    30,
    25,
)  # can/should be increased if running on GPU, or if extra patient
grid = hj.Grid.from_lattice_parameters_and_boundary_conditions(
    state_domain, grid_resolution, periodic_dims=2
)
```

```
target_values = hj.utils.multivmap(target_set, np.arange(4))(grid.states)
envelope_values = hj.utils.multivmap(envelope_set, np.arange(4))(grid.states)
terminal_values = np.maximum(target_values, envelope_values)

solver_settings = hj.SolverSettings.with_accuracy(
    "medium",  # can/should be changed to "very_high" if running on GPU, or if extra patient
    hamiltonian_postprocessor=lambda x: jnp.minimum(x, 0),
    value_postprocessor=lambda t, x: jnp.maximum(x, envelope_values),
)

# Propagate the HJ PDE _backwards_ in time.
initial_time = 0.0
final_time = -5.0
yn = None
if os.path.exists("hj_reachability_values.npz"):
    yn = (
        input(
            "Existing hj_reachability_values.npz file found from a previous solve; use it
(Y/n)? "
        )
        .lower()
        .strip()
    )
if yn is None or yn == "n":
    print("Computing the value function by solving the HJ PDE.")
    values = hj.step(
        solver_settings,
        planar_quadrotor,
        grid,
        initial_time,
        terminal_values,
        final_time,
    ).block_until_ready()
    print("Saving the value function to hj_reachability_values.npz.")
    np.savez("hj_reachability_values.npz", values=values)
else:
    print("Loading previously computed value function from hj_reachability_values.npz.")
    values = np.load("hj_reachability_values.npz")["values"]
grad_values = grid.grad_values(values)

# Utilities for rolling out the optimal controls and visualizing.


@jax.jit
def optimal_step(full_state, dt):
    state = full_state[2:]
    grad_value = grid.interpolate(grad_values, state)
    control = planar_quadrotor.optimal_control(state, grad_value)
    return full_state + dt * planar_quadrotor.full_dynamics(full_state, control)


def optimal_trajectory(full_state, dt=1 / 100, T=5):
    full_states = [full_state]
    t = np.arange(T / dt) * dt
    for _ in t:
        full_states.append(optimal_step(full_states[-1], dt))
    return t, np.array(full_states)


def animate_optimal_trajectory(full_state, dt=1 / 100, T=5, display_in_notebook=False):
    t, full_states = optimal_trajectory(full_state, dt, T)
    value = grid.interpolate(values, full_state[2:])
    fig, anim = animate_planar_quad(
        t,
        full_states[:, 0],
        full_states[:, 2],
        full_states[:, 4],
```

```
        #f"V = {value:7.4f}",
        #display_in_notebook=display_in_notebook,
    )
    return fig, anim


# Dropping the quad straight down (v_y = -5, mimicking waiting for a sec after the drop to turn
the props on).
state = [5.0, -5.0, 0.0, 0.0]
fig, ani = animate_optimal_trajectory(np.array([0, 0] + state))
ani.save("planar_quad_1.mp4", writer="ffmpeg")
plt.show()

# Flipping the quad up into the air.
state = [6.0, 2.0, -3 * np.pi / 4, -4.0]
fig, ani = animate_optimal_trajectory(np.array([0, 0] + state))
ani.save("planar_quad_2.mp4", writer="ffmpeg")
plt.show()

# Dropping the quad like a falling leaf.
state = [8.0, -0.8, np.pi / 2, 2.0]
fig, ani = animate_optimal_trajectory(np.array([0, 0] + state))
ani.save("planar_quad_3.mp4", writer="ffmpeg")
plt.show()

# Too much negative vertical velocity to recover before hitting the floor.
state = [8.0, -3.0, np.pi / 2, 2.0]
fig, ani = animate_optimal_trajectory(np.array([0, 0] + state))
ani.save("planar_quad_4.mp4", writer="ffmpeg")
plt.show()

# Examining an isosurface (exercise part (d)).
import plotly.graph_objects as go

i_y = 18
fig = go.Figure(
    data=go.Isosurface(
        x=grid.states[i_y, ..., 1].ravel(),
        y=grid.states[i_y, ..., 2].ravel(),
        z=grid.states[i_y, ..., 3].ravel(),
        value=values[i_y].ravel(),
        colorscale="jet",
        isomin=0,
        surface_count=1,
        isomax=0,
    ),
    layout_title=f"Zero isosurface, y = {grid.coordinate_vectors[0][i_y]:7.4f}",
    layout_scene_xaxis_title="v_y",
    layout_scene_yaxis_title="phi",
    layout_scene_zaxis_title="omega",
)
#fig.write_image("isosurface.png")
fig.show()
```

Problem 3.3: MPC Feasibility

a) Discrete LTI system: $x_{k+1} = Ax_k + Bu_k$

Receding Horizon Controller for Quadratic cost J: $J(x,u) = x_T^T P x_T + \sum_{k=0}^{T-1} (x_k^T Q x_k + u_k^T R u_k)$, $P, Q, R \succ 0$ are weight matrices

Satisfy state and input constraints: $\|x_k\|_\infty \leq c_x$, $\|u_k\|_\infty \leq c_u$

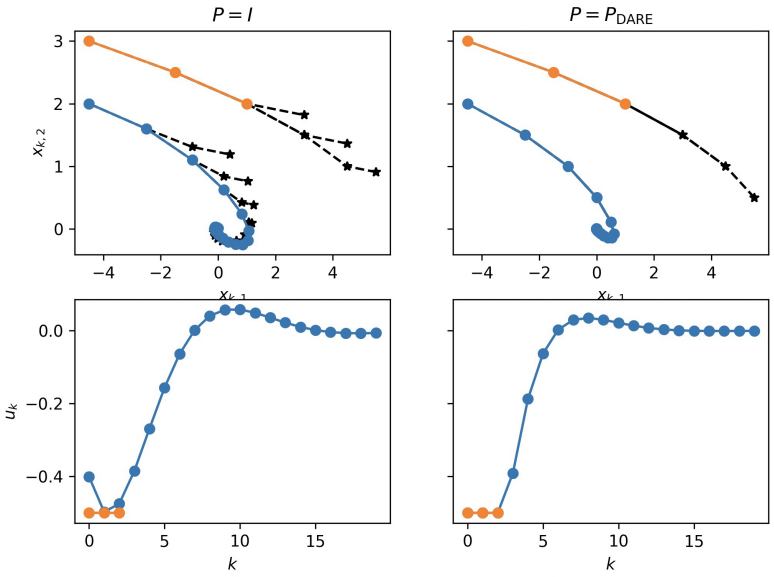
Terminal state constraint: $\|x_T\|_\infty \leq c_T$, tune $c_T \geq 0$

For $c_T = 0$, terminal state constraint is equivalent to $x_T = 0$, $c_T \geq c_x$ gives $\|x_T\|_\infty \leq c_x$

Implement receding horizon controller with cvxpy for $c_T \geq c_x$ with $P = I$, or P for Ricatti Eqn:

$$A^TPA - P - A^TPB(R + B^TPB)^{-1}B^TPA + Q = 0$$

See attached code for the implementation of dlqmpc



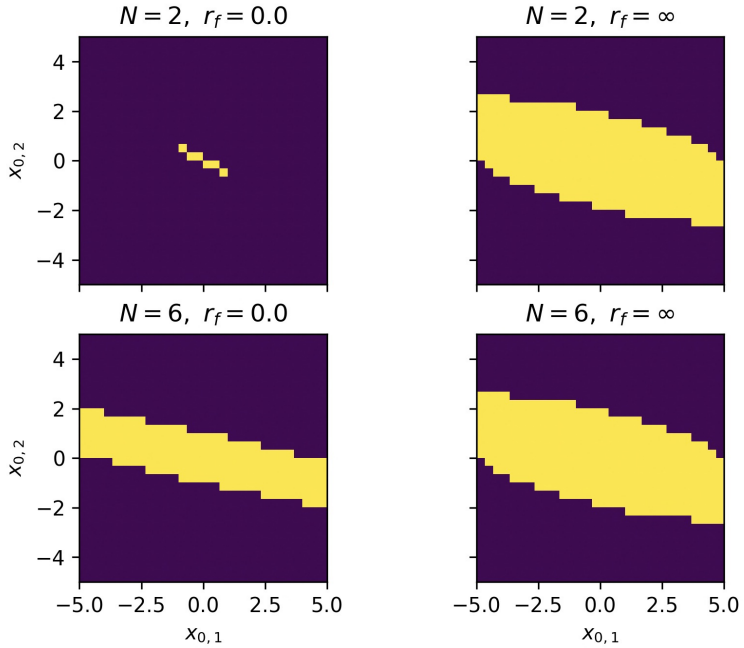
When using $P = P_{DARE}$ when compared to $P = I$, the system is more stable noted by the more convenient predictions in x_1. The control sequence is also slightly smoother when using P_{DARE}

b) Compute the region of attraction (ROA) for fixed $P \succ 0$ and different values of N and r_f

ROA is the set of initial states from which the system, under a specific control law (like MPC), will asymptotically converge to a eqm point (origin) without violating any constraints

For every square in the grid (initial condition), iterate for up to max_steps to determine if we ever reach the origin using mpc. If we do before max_steps, roa[i,j]= 1, and break. If cvxpy ever returns infeasible, roa[i,j]=0 and break. Find the new x based on $x = Ax + B u_{mpc}[0]$ since we only want the first control from mpc's sequence.

See attached code for implementation of compute_roa.



When using a more constraining r_f ($r_f < \infty$), the region of attraction becomes smaller because there are less possible initial conditions to meet the tighter constraint. Also, a smaller N leads to a similar result, where a larger N means a larger region of attraction.

```python
"""
Starter code for the problem "MPC feasibility".

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

from itertools import product

import cvxpy as cvx

import matplotlib.pyplot as plt

import numpy as np

from scipy.linalg import solve_discrete_are

from tqdm.auto import tqdm


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,
) -> tuple[np.ndarray, np.ndarray, str]:
    """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.

    cost = 0.0
    constraints = [x_cvx[0] == x0]

    for t in range(N):
        cost += cvx.quad_form(x_cvx[t], Q) + cvx.quad_form(u_cvx[t], R)
        constraints += [x_cvx[t + 1] == A @ x_cvx[t] + B @ u_cvx[t]]
        constraints += [cvx.norm(x_cvx[t], 'inf') <= rx]
        constraints += [cvx.norm(u_cvx[t], 'inf') <= ru]

    cost += cvx.quad_form(x_cvx[N], P)
    constraints += [cvx.norm(x_cvx[N], 'inf') <= rf]

    # 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


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 = 30,
    max_steps: int = 20,
    tol: float = 1e-2,
) -> np.ndarray:
    """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`.

            # END PART (b) ########################################################
            converged = False
            for _ in range(max_steps):
                x_mpc, u_mpc, status = do_mpc(x, A, B, P, Q, R, N, rx, ru, rf)
                if status == "infeasible":
                    break
                x = A @ x + B @ u_mpc[0]
                if np.linalg.norm(x, np.inf) < tol:
                    converged = True
                    break
            if converged:
                roa[i, j] = 1
    return roa


# Part (a): Simulate and plot trajectories of the closed-loop system
n, m = 2, 1
A = np.array([[1.0, 1.0], [0.0, 1.0]])
B = np.array([[0.0], [1.0]])
Q = np.eye(n)
R = 10.0 * np.eye(m)
P_dare = solve_discrete_are(A, B, Q, R)
N = 3
T = 20
rx = 5.0
ru = 0.5
rf = np.inf

Ps = (np.eye(n), P_dare)
titles = (r"$P = I$", r"$P = P_\mathrm{DARE}$")
x0s = (np.array([-4.5, 2.0]), np.array([-4.5, 3.0]))

fig, ax = plt.subplots(2, len(Ps), dpi=150, figsize=(10, 8), sharex="row", sharey="row")
for i, (P, title) in enumerate(zip(Ps, titles)):
    for x0 in x0s:
        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(x, A, B, P, Q, R, N, rx, ru, rf)
            if status == "infeasible":
                x_mpc = x_mpc[:t]
                u_mpc = u_mpc[:t]
                break
            x = A @ x + B @ u_mpc[t, 0, :]
```

```
            ax[0, i].plot(x_mpc[t, :, 0], x_mpc[t, :, 1], "--*", color="k")
        ax[0, i].plot(x_mpc[:, 0, 0], x_mpc[:, 0, 1], "-o")
        ax[1, i].plot(u_mpc[:, 0], "-o")
    ax[0, i].set_title(title)
    ax[0, i].set_xlabel(r"$x_{k,1}$")
    ax[1, i].set_xlabel(r"$k$")
ax[0, 0].set_ylabel(r"$x_{k,2}$")
ax[1, 0].set_ylabel(r"$u_k$")
fig.savefig("mpc_feasibility_sim.png", bbox_inches="tight")
plt.show()


# Part (b): Compute and plot regions of attraction for different MPC parameters
print("Computing regions of attraction (this may take a while) ... ", flush=True)
Ns = (2, 6)
rfs = (0.0, np.inf)
fig, axes = plt.subplots(
    len(Ns), len(rfs), dpi=150, figsize=(10, 10), sharex=True, sharey=True
)
prog_bar = tqdm(product(Ns, rfs), total=len(Ns) * len(rfs))
for flat_idx, (N, rf) in enumerate(prog_bar):
    i, j = np.unravel_index(flat_idx, (len(Ns), len(rfs)))
    roa = compute_roa(A, B, P_dare, Q, R, N, rx, ru, rf, grid_dim=30)
    axes[i, j].imshow(
        roa.T, origin="lower", extent=[-rx, rx, -rx, rx], interpolation="none"
    )
    axes[i, j].set_title(
        r"$N = {},\ r_f = $".format(N) + (r"$\infty$" if rf == np.inf else str(rf))
    )
for ax in axes[-1, :]:
    ax.set_xlabel(r"$x_{0,1}$")
for ax in axes[:, 0]:
    ax.set_ylabel(r"$x_{0,2}$")
fig.savefig("mpc_feasibility_roa.png", bbox_inches="tight")
plt.show()
```

Problem 3.4: Terminal Ingredients

a) DT LTI system: $x_{t+1} = Ax_t + Bu_t$, $A = \begin{bmatrix} 0.9 & 0.6 \\ 0 & 0.8 \end{bmatrix}$, $B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

Synthesize MPC to regulate the system to the origin while minimizing cost:

$$J(x,u) = x_N^T P x_N + \sum_{t=0}^{N-1} (x_t^T Q x_t + u_t^T R u_t), \quad Q \succ 0, \ R \succ 0, \ P \succ 0$$

subject to $\|u_t\|_\infty \le r_u$, $\|u_t\|_\infty \le r_u$, $x_T \in \mathcal{X}_T$

Let $N = 5$, $r_x = 5$, $r_u = 1$, $Q = I$, $R = I$

How to design $\mathcal{X}_T$ and P in an open-loop manner, by only considering system $x_{t+1} = Ax_t$

We are able to only look at the open-loop dynamics $x_{t+1} = Ax_t$ when designing the terminal ingredients (the terminal set $\mathcal{X}_f$ and terminal cost P) under conditions where the system dynamics are well-understood and the constraints are straightforward.

↳ The system is open loop stable if the eigenvalues of the matrix A are within the unit circle ← the system naturally tends to 0 over time

- We have $A = \begin{bmatrix} 0.9 & 0.6 \\ 0 & 0.8 \end{bmatrix}$ ← $\lambda(A) = 0.8, 0.9$
- Since the eigenvalues of A are less than 1 in magnitude, the system is open-loop stable

Since we know that this system is inherently open-loop stable, we can design $\mathcal{X}_T$ and P based on this.

First, we can derive P via the DARE equation. Designing P without the control input reflects that the system is open-loop stable

↳ The solution to the DARE will be the Lyapunov Control Function that drives the state towards the origin. P then will decrease along the trajectories of the closed-loop system

To ensure recursive feasibility, we must ensure that the terminal set $\mathcal{X}_T$ is an invariant set under the terminal control law

↳ To do this, we must ensure that $\|x\|_\infty \le r_x$ under the dynamics $x_{t+1} = Ax_t$

↳ We must ensure that for all $x \in \mathcal{X}_f$, the state remains in $\mathcal{X}_T$ under the dynamics $x_{t+1} = Ax_t$
(↑ we did this by checking $\lambda(A)$)

By designing P using DARE, and ensuring $\mathcal{X}_T$ is an invariant set within the state constraints, we can guarantee that the terminal ingredients satisfy the properties needed for recursive feasibility and the stability of the system with MPC feedback.

b) Find the largest positive invariant set $\mathcal{X}_f$ for $x_{t+1} = Ax_t$ that satisfies state constraints

Search for ellipsoids of the form

$$\mathcal{X}_T = \{x \in \mathbb{R}^n \mid x^T W x \le 1\} \quad \text{with } W \succ 0$$

Since $\text{vol}(\mathcal{X}_T) \sim \sqrt{\det(W^{-1})}$, formulate the search as the SDP

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

Each constraint is a convex or concave SDP is an LMI

Prove that $A^TWA - W \preceq 0$, $I - r_x^2 W \preceq 0$ are sufficient conditions for $\mathcal{X}_T$ to be a positive invariant set satisfying state constraints

2 parts:

- Positive Invariance: If $x_t \in \mathcal{X}_T$, then $x_{t+1} = Ax_t \in \mathcal{X}_T$

$\mathcal{X}_T = \{x \in \mathbb{R}^n \mid x^T W x \le 1\}$

We need to show that $x_{t+1}^T W x_{t+1} \le 1$ given that $x_t^T W x_t \le 1$ ← $x_{t+1} = Ax_t$

$$(Ax_t)^T W A x_t = x_t^T A W A x_t$$

In order for $\mathcal{X}_T$ to be positive invariant, we must then require that

$$x_t^T A^T W A x_t \le x^T W x_t$$

$$x_t^T A^T W A x_t - x^T W x_t \le 0$$

$$x_t^T (A^T W A - W) x_t \le 0$$

The condition $x_t^T (A^TWA - W) x_t \le 0$ is equivalent to having the constraint that

$$(A^TWA - W) \preceq 0 \quad \text{since the above inequality must be true for all } x_t \in \mathbb{R}^n$$

- State Constraints:

To satisfy state constraints, we must ensure that $\|x\|_\infty \le r_x$

↳ This is equivalent to $\|x\|_\infty^2 = x^T x \le r_x^2$ in squared terms

Since we know that $x^T W x \le 1$ from $\mathcal{X}_T$, we can also write a more constraining inequality:

$$x^T I x \le r_x^2 x^T W x \quad \leftarrow \text{since } r_x^2 \le r_x^2 x^T W x \text{ must be true}$$

$$x^T I x - r_x^2 x^T W x \le 0 \quad \leftarrow r_x^2 \text{ is a scalar}$$

$x^T(I - r_w^2 W)x \leq 0$

$\therefore$ This is equivalent the condition that $I - r_w^2 W \succeq 0$

By this analysis, satisfying $A^T W A - W \preceq 0$ and $I - r_w^2 W \succeq 0$ are sufficient conditions to ensure that $\mathcal{X}_T$ is a positive invariant set that satisfies state constraints.

c) Maximization problems → we want concave fns

This SDP is not concave since $\log\det(W^{-1}) = -\log\det(W)$ is convex wrt the argument $W \succ 0$

Reformulate the SDP as a concave in $M = W^{-1}$

$\log\det(W^{-1})$ is not concave

$\log\det(W^{-1}) = -\log\det(W)$ ← maximizing $\log\det(W^{-1})$ is equivalent to minimizing $\log\det(W)$

↳ change variable to M, $M = W^{-1}$

$\therefore$ Our objective becomes minimize $\log\det(M)$

$A^T W A - W \preceq 0$

$A^T M^{-1} A - M^{-1} \preceq 0$

$M(A^T M^{-1} A - M^{-1})M \preceq M \cdot 0 \cdot M$

$MA^T M^{-1} AM - MM^{-1}M \preceq 0$

$MA^T M^{-1} AM - M \preceq 0$

$AMA^T - M \preceq 0$

$I - r_w^2 W \succeq 0$

$I - r_w^2 M^{-1} \succeq 0$

$I \preceq r_w^2 M^{-1}$

$\frac{1}{r_w^2} I \preceq M^{-1}$ ← By second hint, $A \preceq \gamma I$ if and only if $A^{-1} \succeq \frac{1}{\gamma} I$

$\therefore M \preceq \frac{1}{r_w^2} I$

Concave SDP Problem:

maximise $\log\det(M)$

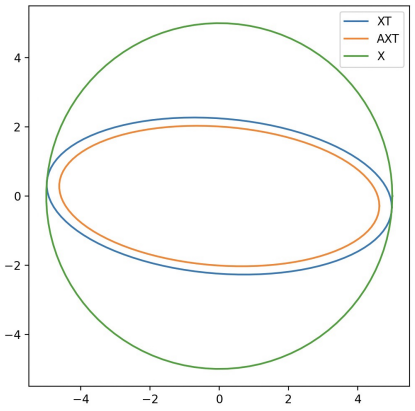
subject to $AMA^T - M \preceq 0$

$M \preceq \frac{1}{r_w^2} I$

d) Solve SDP for M using NP and CVX

See attached code for implementation of part d

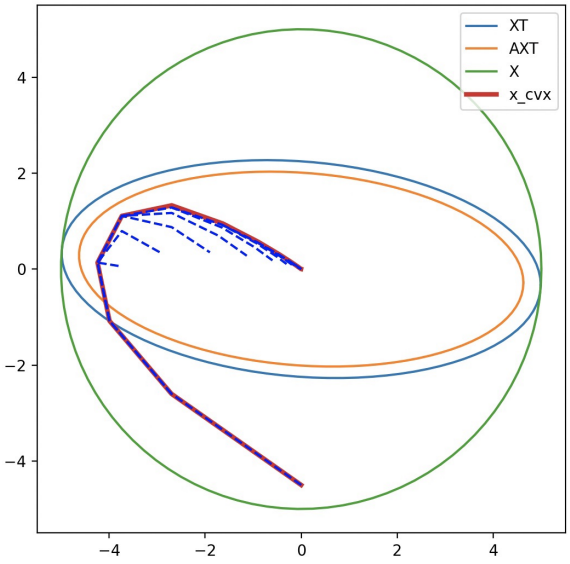
Plot of ellipsoids:



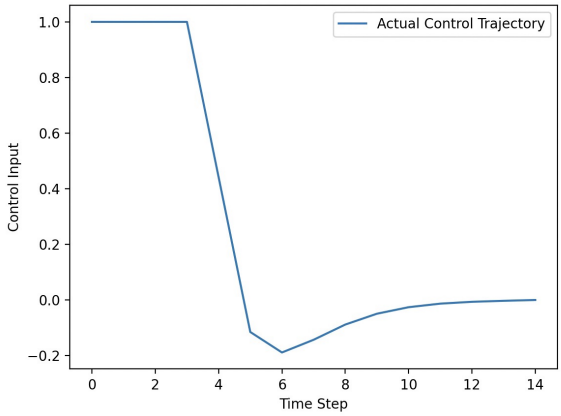
$W = M^{-1} =$

```
[[0.0410825  0.01308029]
 [0.01308029 0.19805235]]
```

e) Overlayed Plot with actual state trajectory and planned trajectories



Actual Control Trajectory:



See attached code for implementation

```python
import numpy as np
import cvxpy as cvx
import matplotlib.pyplot as plt

##### Part D #####
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

# Define the parameters
A = np.array([[0.9, 0.6], [0.0, 0.8]])
N = 4
B = np.array([[0], [1]])
Q = np.eye(2)
R = np.eye(1)
P = np.eye(2)
rx = 5
ru = 1
N = 4
x0 = np.array([0, -4.5])

M = cvx.Variable((2, 2), symmetric=True) # Ensure M is symmetric
objective = cvx.Maximize(cvx.log_det(M))
constraints = [
    M >> 0,
    cvx.quad_form(A.T, M) - M << 0,
    M << (rx**2) * np.eye(2)
]

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

status = prob.status
print(status)
M_opt = M.value
W_opt = np.linalg.inv(M_opt)
print(W_opt)

# Generate points for ellipsoids XT and AXT
XT_points = generate_ellipsoid_points(M_opt)
AXT_points = generate_ellipsoid_points(A @ M_opt)

# Generate points for X
r = np.sqrt(rx**2)
X_points = generate_ellipsoid_points(r**2 * np.eye(2))

# Plot the ellipsoids
fig, ax = plt.subplots()
ax.plot(XT_points[:, 0], XT_points[:, 1], label='XT')
ax.plot(AXT_points[:, 0], AXT_points[:, 1], label='AXT')
ax.plot(X_points[:, 0], X_points[:, 1], label='X')
ax.set_aspect('equal')
ax.legend()

##### Part E #####

##### Find Actual Trajectory #####
t_steps = 15
x = cvx.Variable((2, t_steps+1))
u = cvx.Variable((1, t_steps))
n, m = Q.shape[0], R.shape[0]

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

```
cost = 0.0
constraints = [x_cvx[0] == x0]

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

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

# Solve the MPC problem
prob = cvx.Problem(cvx.Minimize(cost), constraints)
prob.solve()

# Extract the optimal values
x_opt = x_cvx.value
u_opt = u_cvx.value

# Plot the state trajectory
ax.plot(x_opt[:, 0], x_opt[:, 1], label='x_cvx', linewidth=3.0)
ax.set_aspect('equal')
ax.legend()

##### Find the Planned Trajectories #####
def solve_mpc(x0):
    x = cvx.Variable((2, N+1))
    u = cvx.Variable((1, N))
    cost = 0
    constraints = [x[:, 0] == x0]
    for t in range(N):
        cost += cvx.quad_form(x[:, t], Q) + cvx.quad_form(u[:, t], R)
        constraints += [x[:, t+1] == A @ x[:, t] + B @ u[:, t],
                        cvx.norm(x[:, t+1], 2) <= rx,
                        cvx.norm(u[:, t], 2) <= ru]
    cost += cvx.quad_form(x[:, N], P)
    problem = cvx.Problem(cvx.Minimize(cost), constraints)
    problem.solve()
    return x.value, u.value

x_traj = [x0]
u_traj = []

for t in range(t_steps):
    x0 = x_traj[-1]
    x_pred, u_pred = solve_mpc(x0)
    x_traj.append(x_pred[:, 1])
    u_traj.append(u_pred[:, 0])

x_traj = np.array(x_traj)
u_traj = np.array(u_traj)

# Overlay the planned trajectories at each time step
for t in range(t_steps):
    x0 = x_traj[t]
    x_pred, _ = solve_mpc(x0)
    ax.plot(x_pred[0, :], x_pred[1, :], 'b--')

plt.show()

# Plot the actual control trajectory
fig, ax = plt.subplots()
ax.plot(range(t_steps), u_opt, label='Actual Control Trajectory')
ax.set_xlabel('Time Step')
ax.set_ylabel('Control Input')
ax.legend()
```

```
plt.show()
```