Stanford **Spring 2024**

AA 203: Optimal and Learning-based Control
Homework #3
Due May 20 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.

(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.

let $x = (x_1, x_2)$

given a, spse x' is the next state specified by action a

$$p(x'; x, a) = (1 - w(x)) + 0.25\, w(x)$$

it will move into any other state x_i' that does not correspond to the action a

$$p(x_i'; x, a) = 0.25\, w(x) \quad i \in [1, 2, 3]$$

The value @ the final grid point $x_{goal} = (19, 9)$ is

$$V(x_{goal}) = R(x_{goal}) = I_{x_{goal}}(x_{goal}) = 1$$

For example, at $(19, 8)$

if a = up, $x' = (19, 9)$ $\quad p(x'; x, a) = (1 - w(x)) + 0.25\, w(x)$

$$R(x') + \gamma V(x') = 1 + \gamma$$

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__`).

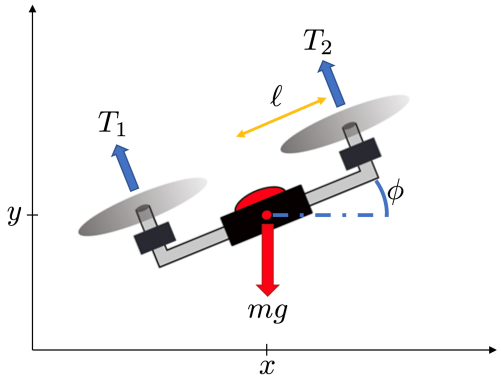


Figure 1: A planar quadrotor.

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).

(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`.

$$H = \begin{bmatrix} p_1 & p_2 & p_3 & p_4 \end{bmatrix} \begin{bmatrix} v_y \\ \frac{(T_1+T_2)\cos\phi - C_D^v v_y}{m} \\ w \\ \frac{(T_2-T_1)l - C_D^\phi w}{I_{yy}} \end{bmatrix}$$

$$= p_1 v_y + p_2 \frac{(T_1+T_2)\cos\phi - C_D^v v_y}{m} + p_3 w + p_4 \frac{(T_2-T_1)l - C_D^\phi w}{I_{yy}}$$

$$\operatorname*{argmin}_u H = \operatorname*{argmin}_u p_2 \frac{(T_1+T_2)\cos\phi - C_D^v v_y}{m} + p_4 \frac{(T_2-T_1)l - C_D^\phi w}{I_{yy}}$$

$$= \operatorname*{argmin}_u \frac{p_2 \cos\phi}{m}(T_1+T_2) + \frac{p_4}{I_{yy}}(T_2-T_1)$$

this is a linear function in both direction T_1 & T_2

the control limits would cause out a square space

because of linearity, it suffices to check the four corner boundary values and use the minimal as local optimal control

(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.

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

For each state, a function of the form

$$(s - \text{lower\_bound}) \times (s - \text{upper\_bound})$$

is defined.
Then, we take the maximum of the four functions as the output of the target set function

(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`.

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

(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.

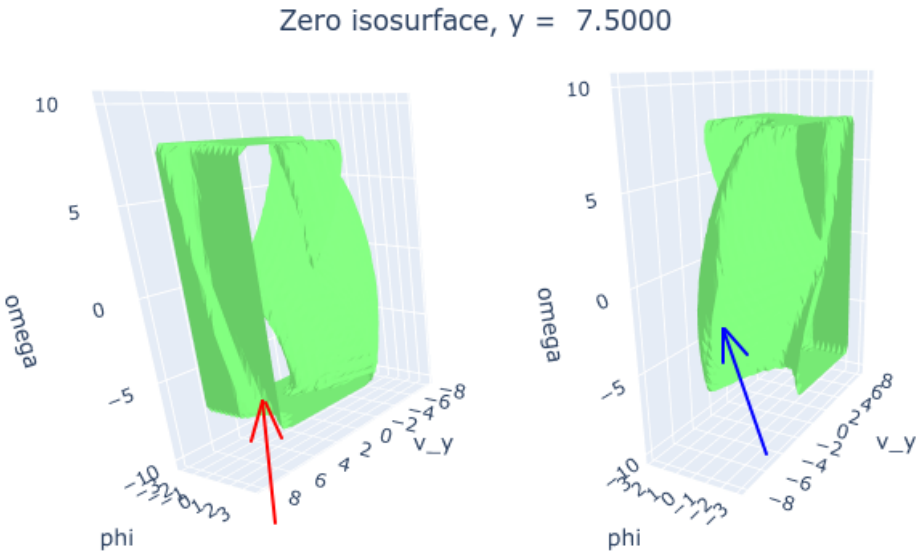


Figure 2: Example zero isosurface views. Can you explain why the red valley (outside the isosurface, i.e., unrecoverable initial conditions) or blue ridge (inside the isosurface, i.e., initial conditions that can reach the target collision-free) exhibit the "tilt" they have by considering the corresponding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

For negative v_y, there is a protrude ridge around $\phi \approx 0$, because, with a faster downward velocity, if our drone is not already in a relatively upright state, there would be very limited time to enact control to bring the drone upright before the drone hit the ground

Similarly,
for positive v_y, there is a concave ridge around $\phi \approx 0$ because if $\phi \approx 0$. i.e. the drone is upright to begin with, there is a good chance that it will fly out of the upper bound with additional thrusts on the rotors.

(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.

Pro of DP against MPC:

- Ensure optimality/reachability of target set over long horizon and the worst possible adversaries instead of finding optimal solution for a limited horizon and resulting in potentially less desirable outcomes in the long run

Con of DP against MPC:

- less reactive modeling inaccuracy or disturbance
- cannot be computed in near real time to account for environment changes as the PDE solver is rather complicated

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`.

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.

Since we are interested in driving the discrete system to zero, we evaluate the eigenvalues of state transition matrix and find them to be less than 1

thus, this system is open loop stable.

Therefore, we can simply derive the set $\mathcal{X}_T$ and the penalty P based on open loop dynamics as we can always reach the desired terminal state $[0, 0]^T$ simply by applying no control.

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.

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.

$$x^\mathsf{T} A^\mathsf{T} W A x - x^\mathsf{T} W x \leq 0$$

$$x_{k+1}^\mathsf{T} W x_{k+1} - x_k^\mathsf{T} W x_k \leq 0$$

$$x_{k+1}^\mathsf{T} W x_{k+1} \leq x_k^\mathsf{T} W x_k \leq 1$$

$$I - r_x^2 W \preceq 0$$

$$x_k^\mathsf{T} I x_k - r_x^2 x_k^\mathsf{T} W x_k \leq 0$$

$$\|x_k\|^2 \leq r_x^2 x_k^\mathsf{T} W x_k \leq r_x^2 1$$

- A set $C \subseteq X$ is said to be a control invariant set for the system $\mathbf{x}(t+1) = \phi\big(\mathbf{x}(t), \mathbf{u}(t)\big)$ with constraints $\mathbf{x}(t) \in X$, $\mathbf{u}(t) \in U$, if:

$$\mathbf{x}(t) \in C \Rightarrow \exists \mathbf{u} \in U \text{ such that } \phi\big(\mathbf{x}(t), \mathbf{u}(t)\big) \in C, \text{ for all } t$$

(b) Prove that $A^\mathsf{T}WA - W \preceq 0$ 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.

We want to show that
if $A^TWA - W \preceq 0$ and $I - r^2W \preceq 0$

$\mathcal{X}_T = \{x \in \mathbb{R}^n \mid x^TWx \le 1\}$ is a positive invariant set. that is,

for $x_0 \in \mathcal{X}_T$, $x_k \in \mathcal{X}_T$ for all k with $x_0 = x_0$

Spse we have arbitrary $x_k \in \mathcal{X}_T$, that is $x_k W x_k \le 1$

$$A^TWA - W \preceq 0$$
$$x_k^TA^TWA^Tx_k - x_k^TWx_k \le 0$$
$$x_{k+1}^TWx_{k+1} \le x_kWx_k$$

thus $x_{k+1}^TWx_{k+1} \le 1$, $x_{k+1} \in \mathcal{X}_T$

thus $x_k \in \mathcal{X}_T$ means $x_{k+1} \in \mathcal{X}_T$

At the same time, we want to make sure the constraint $\|x_k\|_2 \le r_k$ is satisfies

spse $I - r_x^2 W \preceq 0$ is met

$$x_k^TIx_k - r_x^2 x_k^TWx_k \le 0$$

$$\|x_k\|_2^2 \le r_x^2 x_k^TWx_k \le r_x^2 \cdot 1$$

and $\|x_k\| \leq r_x$

therefore, if $A^T W A - W \preceq 0$

& $I - r_x^2 W \preceq 0$ is met

then $\mathcal{X}_T = \{x \in \mathbb{R}^n \mid x^T W x \leq 1\}$ is a positive invariant set that satisfies our state constraints.

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)*.

(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 $M := W^{-1}$.

Hint: You should use

1. • the fact that $B \preceq C$ if and only if $ABA \preceq ACA$ for symmetric A, B, and C where $A \succ 0$,
2. • 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
   • 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 $A \succ 0$.

We want to rewrite in terms of

$$\max_M \log\det(M)$$

with $M = W^{-1}$, which is also positive definite

$M \succeq 0$ is also symmetric

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

$$\iff \begin{bmatrix} M & A \\ A^\mathsf{T} & M^{-1} \end{bmatrix} \succeq 0$$ by Schur's Complement Lemma

Now, to get rid of M^{-1} in the lower right corner we multiply both sides by

$$\begin{bmatrix} I & 0 \\ 0 & M \end{bmatrix} (\,\cdot\,) \begin{bmatrix} I & 0 \\ 0 & M \end{bmatrix}$$

$$\begin{bmatrix} I & 0 \\ 0 & M \end{bmatrix}\begin{bmatrix} M & A \\ A^T & M^{-1} \end{bmatrix}\begin{bmatrix} I & 0 \\ 0 & M \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & M \end{bmatrix}\begin{bmatrix} M & AM \\ A^T & I \end{bmatrix}$$

$$= \begin{bmatrix} M & AM \\ MA^T & M \end{bmatrix} \succeq 0$$

thanks to fact ①, and $\begin{bmatrix} I & 0 \\ 0 & M \end{bmatrix}$ & $\begin{bmatrix} M & A \\ A^T & M^{-1} \end{bmatrix}$ are bothe symmetric matrices

$$I - r_x^2 W \preceq 0$$
$$r_x^2 W \succeq I$$
$$M^{-1} \succeq \frac{1}{r_x^2} I$$
$$M \preceq r_x^2 I \qquad \text{using fact ②}$$

thus, the problem is re-formulated as

$$\max_M \ \log\det(M)$$
$$\text{subject to} \quad M \preceq r_x^2 I$$
$$\begin{bmatrix} M & AM \\ MA^T & M \end{bmatrix} \succeq 0$$

- 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.

Value_Iter

May 19, 2024

[1]:
```
import numpy as np
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
import seaborn as sns
```

[2]:
```
n = 20
nx = n
ny = n
  = 10   # spread of storm
  = 0.95 # Discount Factor
x_eye = np.array([15, 14]) #(y, x)

x_goal = np.array([9, 19]) #(y, x)



grid = np.zeros((ny, nx))

  = lambda x:  np.exp(-np.linalg.norm(np.array(x) - x_eye, 2)**2 / (2* **2))
 _grid = grid
for y in range(grid.shape[0]):
    for x in range(grid.shape[1]):
         _grid[y,x]  =  ([y,x])
 _grid =  _grid # the indices need to be transposed because moving to the right␣
 ↪is actually moving through columns
```

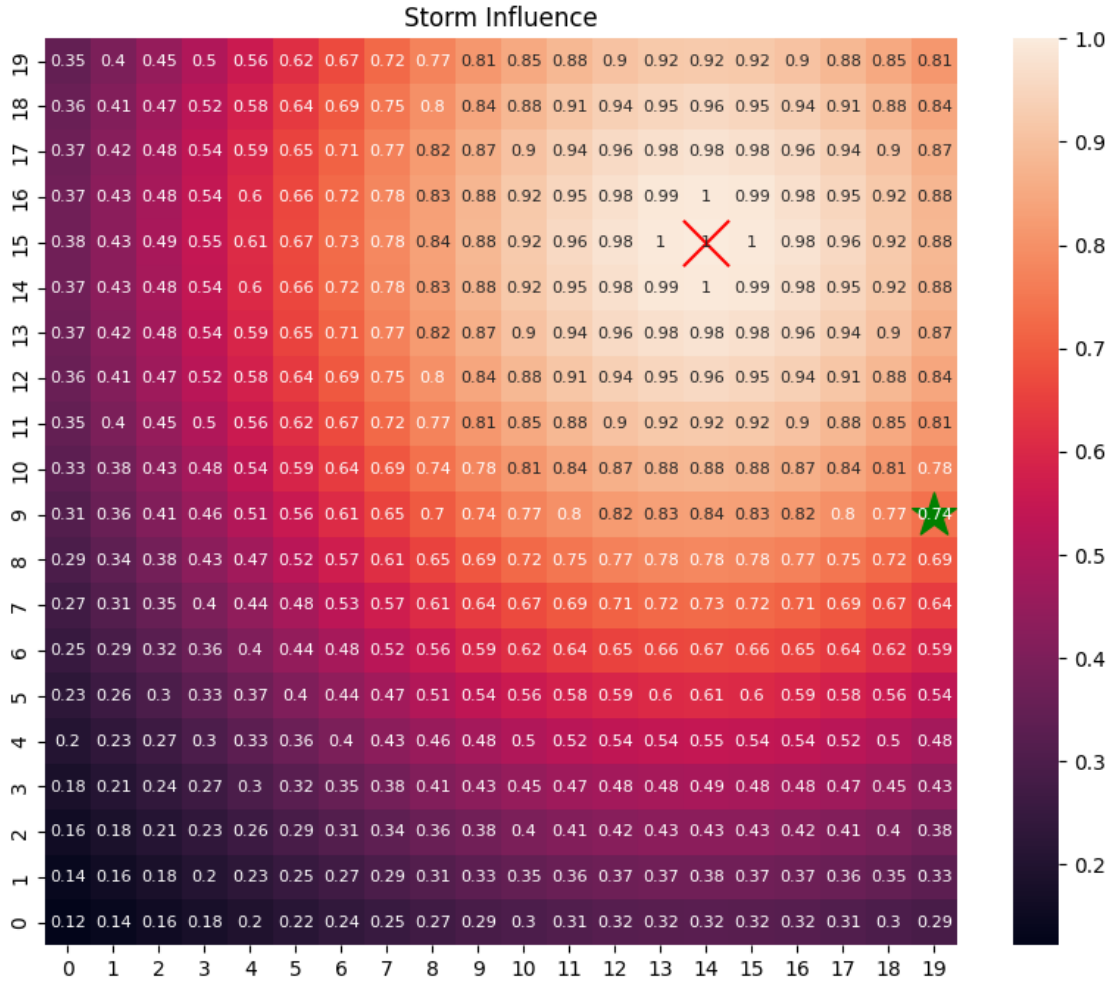
[3]:
```
 _grid[9, 19]
```

[3]:
```
0.7371233743916278
```

[4]:
```
fig, ax = plt.subplots(figsize=(10,8))
sns.heatmap( _grid, annot=True, annot_kws={'fontsize':8}, ax=ax)
ax.scatter(x_goal[1]+0.5,x_goal[0]+0.5,s=500,c='g', marker='*', label='Goal')
ax.scatter(x_eye[1]+0.5,x_eye[0]+0.5,s=500,c='r', marker='x', label='Storm Eye')
ax.invert_yaxis()
ax.set_title('Storm Influence')
```

```
# ax.legend()
```

[4]: Text(0.5, 1.0, 'Storm Influence')



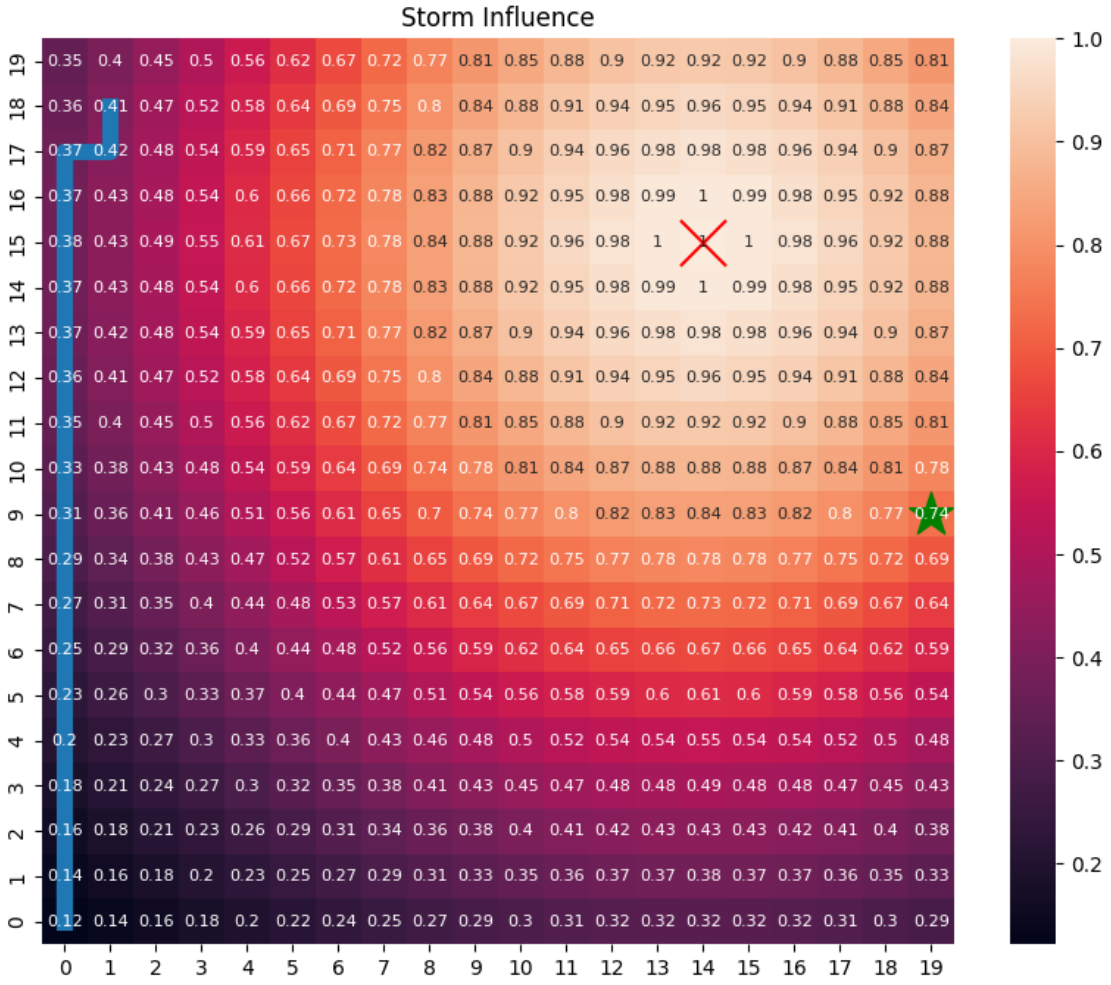
[5]:
```
# define actions
act = {
    3: np.array([0, 1]),     #  right (y, x)
    0: np.array([1, 0]),     #  up (y, x)
    2: np.array([0, -1]),    #  left (y, x)
    1: np.array([-1, 0])     #  down (y, x)
}

def action_dynamics(x:np.array, a:int):
    # x: (y, x)

    if x[0] >= ny or x[1] >= nx:
```

```
        print(x)
        raise ValueError('Outside of Boundary')

    x_next = x + act[a]

    if x_next[0] < 0:
        # print('edge')
        x_next[0] = 0
    elif x_next[0] > ny-1:
        # print('edge')
        x_next[0] = ny-1

    if x_next[1] < 0:
        # print('edge')
        x_next[1] = 0
    elif x_next[1] > nx-1:
        # print('edge')
        x_next[1] = nx-1

    return x_next

# define dynamic rule for each step
def dynamics(x:np.array, a:int,  _grid:np.array, n:int):
    prob = np.random.rand(1)

    # print(x)
    if prob <  _grid[*x]:
        # print('storm')
        a = np.random.randint(0, 4,)

    x_next = action_dynamics(x, a)


    return x_next

# store value iter as 3d vector
```

[8]:
```
x0 = np.array([0, 0])
list_a = [0]*30

x_trj = np.zeros((len(list_a)+1, x0.shape[0]), dtype=int)
x_trj[0] = x0
for i, a in enumerate(list_a):
    x_trj[i+1,:] = dynamics(x_trj[i,:], 0,  _grid, n)
# x_trj
```

[9]:
```
fig, ax = plt.subplots(figsize=(10,8))
sns.heatmap( _grid, annot=True, annot_kws={'fontsize':8}, ax=ax)
ax.scatter(x_goal[1]+0.5,x_goal[0]+0.5,s=500,c='g', marker='*', label='Goal')
ax.scatter(x_eye[1]+0.5,x_eye[0]+0.5,s=500,c='r', marker='x', label='Storm Eye')
ax.plot(x_trj[:,1]+0.5, x_trj[:,0]+0.5, linewidth=8)
ax.invert_yaxis()
ax.set_title('Storm Influence')
# ax.legend()
```

[9]:
```
Text(0.5, 1.0, 'Storm Influence')
```

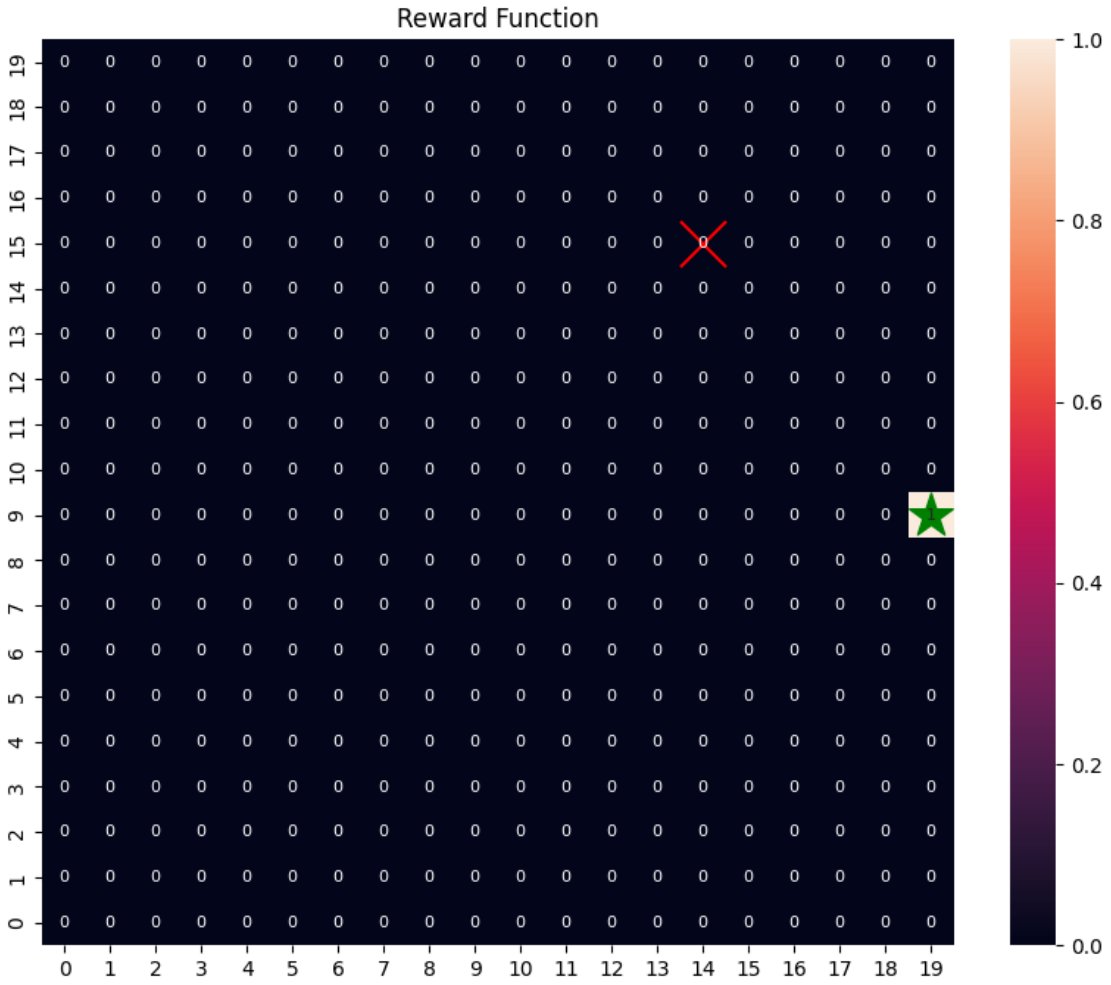


[10]:
```
# _grid[0, 20]
```

[11]:
```
R_grid = np.zeros((ny, nx))
R_grid[*x_goal] = 1
```

```
# R_grid = R_grid.T # the indices need to be transposed because moving to the␣
  ↪right is actually moving through columns
fig, ax = plt.subplots(figsize=(10,8))
sns.heatmap(R_grid, annot=True, annot_kws={'fontsize':8}, ax=ax)
ax.scatter(x_goal[1]+0.5,x_goal[0]+0.5,s=500,c='g', marker='*', label='Goal')
ax.scatter(x_eye[1]+0.5,x_eye[0]+0.5,s=500,c='r', marker='x', label='Storm Eye')
# ax.plot(x_trj[:,0]+0.5, x_trj[:,1]+0.5, linewidth=8)
ax.invert_yaxis()
ax.set_title('Reward Function')
```

[11]: Text(0.5, 1.0, 'Reward Function')

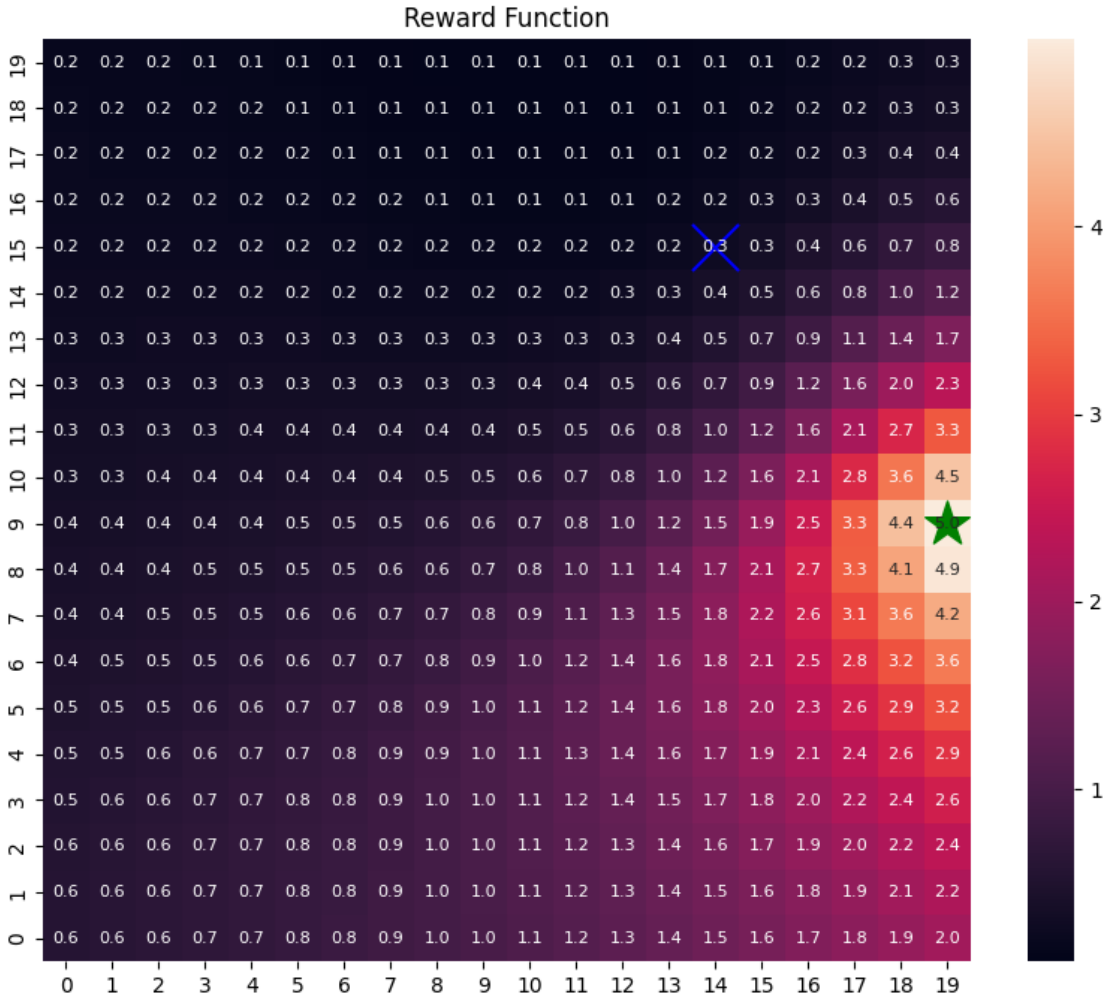


[12]:
```
V_grid = np.zeros(R_grid.shape)
V_grid_next = np.zeros(R_grid.shape)

num_iter = 500
```

```
for _ in range(num_iter):

    V_grid = V_grid_next.copy()

    V_grid_next = np.zeros(R_grid.shape)

    # One iteration of grid update
    for y in range(V_grid.shape[0]):
        for x in range(V_grid.shape[1]):

            list_V = []

            x_neighbors = [action_dynamics(np.array([y, x]), a) for a in act]

            for a in act:

                # update for center locations
                x_next = action_dynamics(np.array([y, x]), a)

                R_plus_V = R_grid[*x_next] +   * V_grid[*x_next]

                prob = (1 -  _grid[y, x])

                sum_val = prob * R_plus_V

                # Account for random movement values
                for x_ngbr in x_neighbors:
                    R_plus_V = R_grid[*x_ngbr] +   * V_grid[*x_ngbr]

                    # equal probability for each of the neighbors based on the␣
 ↪current position
                    prob =  _grid[y, x] / 4

                    sum_val += prob * R_plus_V


                list_V.append(sum_val)
                # print(sum_val)
                # break
            V_grid_next[y, x] = max(list_V)

    if np.max(np.abs(V_grid_next - V_grid)) < 1e-6:
        break
```

```
fig, ax = plt.subplots(figsize=(10,8))
sns.heatmap(V_grid_next, annot=True, annot_kws={'fontsize':8}, ax=ax, fmt='.1f')
ax.scatter(x_goal[1]+0.5,x_goal[0]+0.5,s=500,c='g', marker='*', label='Goal')
ax.scatter(x_eye[1]+0.5,x_eye[0]+0.5,s=500,c='b', marker='x', label='Storm Eye')
# ax.plot(x_trj[:,0]+0.5, x_trj[:,1]+0.5, linewidth=8)
ax.invert_yaxis()
ax.set_title('Reward Function')
```

[12]: Text(0.5, 1.0, 'Reward Function')

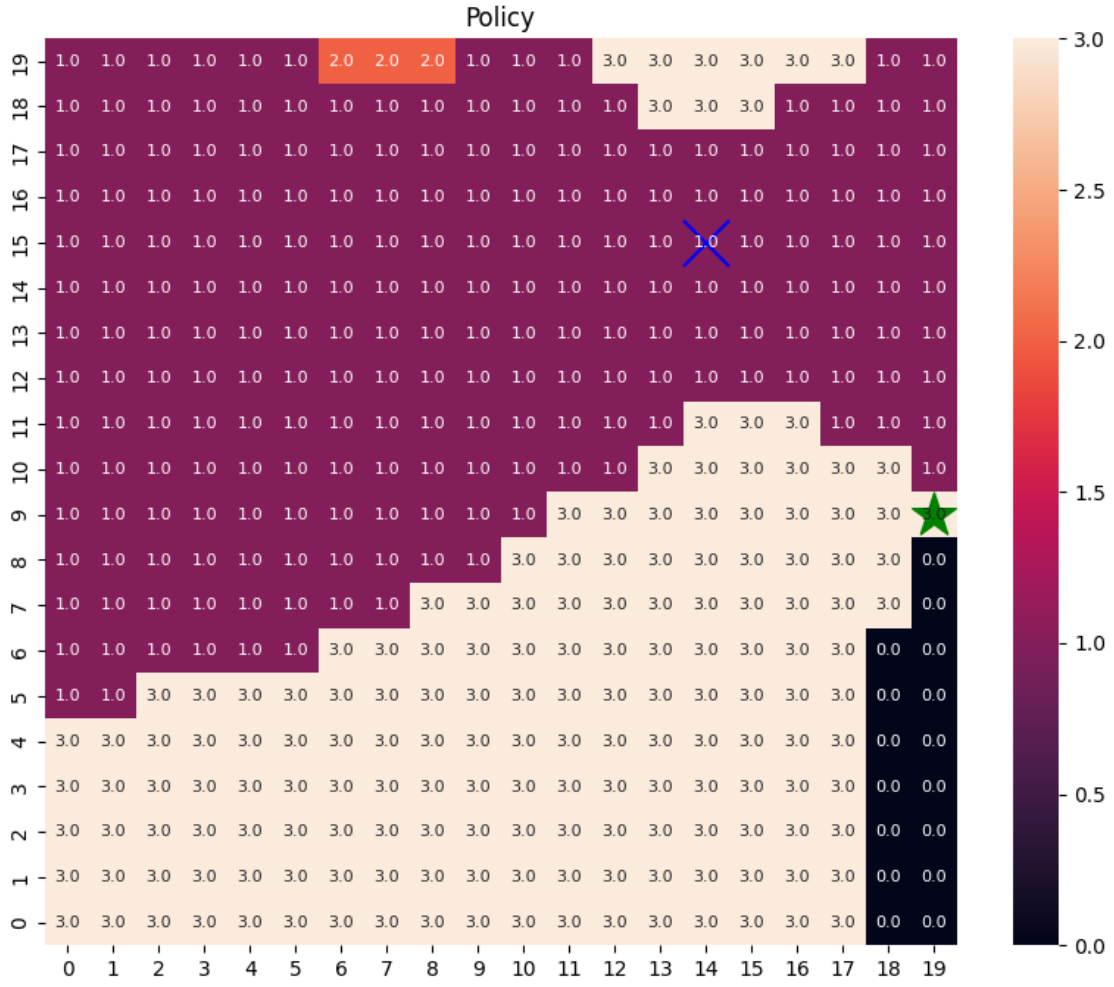


[13]:
```
# Find Optimal Policy based on Value
act_grid = np.ones(R_grid.shape) * -1

for i in range(V_grid.shape[0]):
    for j in range(V_grid.shape[1]):
        # i = 9
```

```
        # j = 15

        a_x_pair = [(a, action_dynamics(np.array([i, j]), a)) for a in act]

        list_a = []
        list_v_diff = []
        for a, x_next in a_x_pair:
            # print(a, x_next)
            V_x = V_grid[i, j]

            V_x_next = V_grid[*x_next]

            list_v_diff.append(V_x_next-V_x)
            list_a.append(a)

        act_grid[i, j] = list_a[np.argmax(list_v_diff)]
        # print(act_grid[i, j], list_v_diff)
    #     break
    # break
    # act_grid[i][j] = np.where(list_v_diff == np.max(list_v_diff))[0]
    # print(np.where(list_v_diff == np.max(list_v_diff))[0])


# act_grid
```

[14]:
```
fig, ax = plt.subplots(figsize=(10,8))
sns.heatmap(act_grid, annot=True, annot_kws={'fontsize':8}, ax=ax, fmt='.1f')
ax.scatter(x_goal[1]+0.5,x_goal[0]+0.5,s=500,c='g', marker='*', label='Goal')
ax.scatter(x_eye[1]+0.5,x_eye[0]+0.5,s=500,c='b', marker='x', label='Storm Eye')
# ax.plot(x_trj[:,0]+0.5, x_trj[:,1]+0.5, linewidth=8)
ax.invert_yaxis()
ax.set_title('Policy')
```

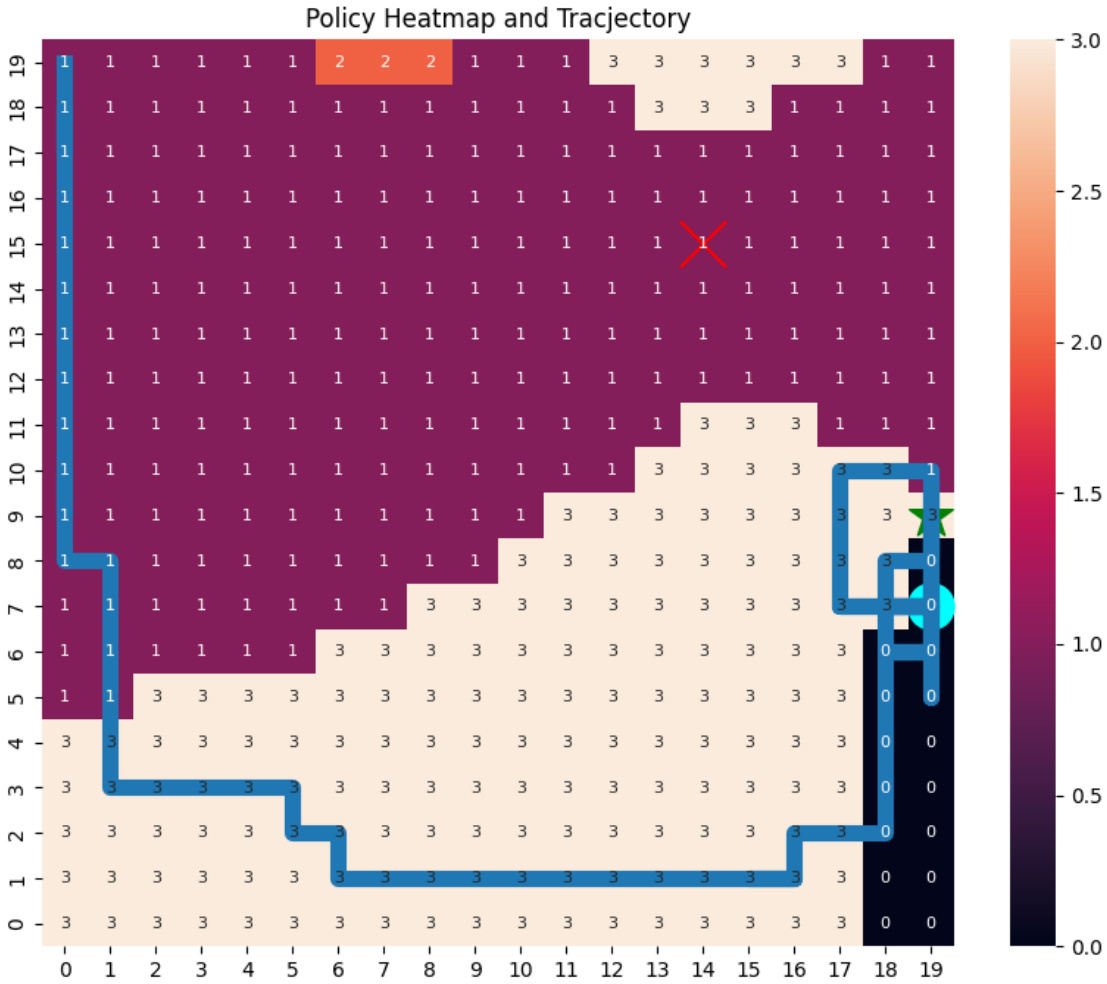
[14]:
```
Text(0.5, 1.0, 'Policy')
```



[17]:
```
x0 = np.array([19, 0]) #y, x

num_iter = 100

x_trj = np.zeros((num_iter+1, x0.shape[0]), dtype=int)
x_trj[0] = x0

for i in range(num_iter):
    a = int(act_grid[*x_trj[i,:]])
    x_trj[i+1,:] = dynamics(x_trj[i,:], a,  _grid, n)

fig, ax = plt.subplots(figsize=(10,8))
sns.heatmap(act_grid, annot=True, annot_kws={'fontsize':8}, ax=ax)
ax.scatter(x_goal[1]+0.5,x_goal[0]+0.5,s=500,c='g', marker='*', label='Goal')
ax.scatter(x_eye[1]+0.5,x_eye[0]+0.5,s=500,c='r', marker='x', label='Storm Eye')
```

```
ax.scatter(x_trj[-1,1]+0.5, x_trj[-1,0]+0.5,s=500,c='cyan', marker='o',␣
 ↪label='Final Destination')
ax.plot(x_trj[:,1]+0.5, x_trj[:,0]+0.5, linewidth=8)
ax.invert_yaxis()
ax.set_title('Policy Heatmap and Tracjectory')
```

[17]: Text(0.5, 1.0, 'Policy Heatmap and Tracjectory')



The policy can be largely divided into three sections.

In the bottom right corner, we are directly below our goal position. The policy simply drives our position upwards until we hit the destination.

In the bottom half of the grid, the policy tells us to keep going right – such that we are either hit right into the goal position, or just go up or down along the grid border until we hit the goal state.

In the top half of the grid, the policy tells us to keep doing down until we reach the region where the optimal policy is to go right until we hit the right edge

[ ]:

hj_reachability

May 19, 2024

[1]:
```
"""
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
```

[2]:
```
9.807 * 2.5 * 0.75
```

[2]:
```
18.388125000000002
```

[3]:
```
# 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
        control = jnp.where(
            jnp.array(
                [
                    jnp.dot(grad_value,  self.dynamics(state, [self.
↪min_thrust_per_prop, self.min_thrust_per_prop])) < \
                    jnp.dot(grad_value,  self.dynamics(state, [self.
↪max_thrust_per_prop, self.min_thrust_per_prop])),
                    jnp.dot(grad_value,  self.dynamics(state, [self.
↪min_thrust_per_prop, self.min_thrust_per_prop])) < \
                    jnp.dot(grad_value,  self.dynamics(state, [self.
↪min_thrust_per_prop, self.max_thrust_per_prop])),
                ]
            ),
            jnp.array([self.min_thrust_per_prop,self.min_thrust_per_prop]),
            jnp.array([self.max_thrust_per_prop,self.max_thrust_per_prop]),
        )
        return 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,
            ]
        )
```

[4]:
```
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
test_optimal_control()
# test_target_set()
# test_envelope_set()
```

```
An NVIDIA GPU may be present on this machine, but a CUDA-enabled jaxlib is not installed. Falling back to cpu.
```

[34]:
```
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
    ################################################
    # raise NotImplementedError
      = [[3, 7], [-1, 1], [-np.pi/12, np.pi/12], [-1, 1]]

    h = jnp.array( [(state[i] -  _s[0])*(state[i] -  _s[1]) for i, _s in enumerate( )] )
```

```
    return jnp.max(h)


␣
↪#############################################################################
```

[35]:
```
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
```

```
test_target_set()
```

[36]:
```
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␣
↪################################################
     = [[1, 9], [-6, 6], [-np.inf, np.inf], [-8, 8]]

    e = jnp.array( [(state[i] - _s[0])*(state[i] - _s[1]) for i, _s in␣
↪enumerate( )] )

    e = jnp.where(jnp.array([True, True, False, True]), e, jnp.array([0., 0.,␣
↪-10., 0.]))

    return jnp.max(e)
  ␣
↪#############################################################################
```

[37]:
```
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_envelope_set()
```

[38]:
```
# 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)
```

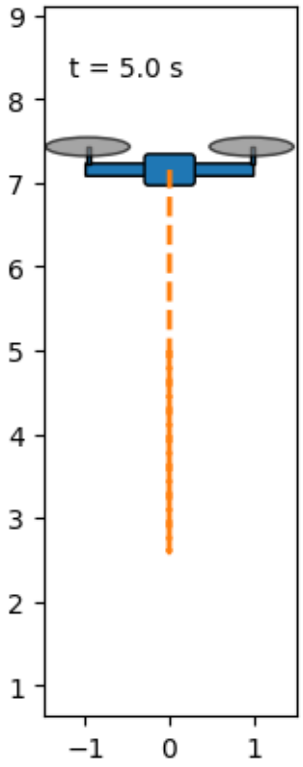
```
Computing the value function by solving the HJ PDE.
100%|##################################|  5.0000/5.0 [21:33<00:00, 258.74s/sim_s]
Saving the value function to hj_reachability_values.npz.
```

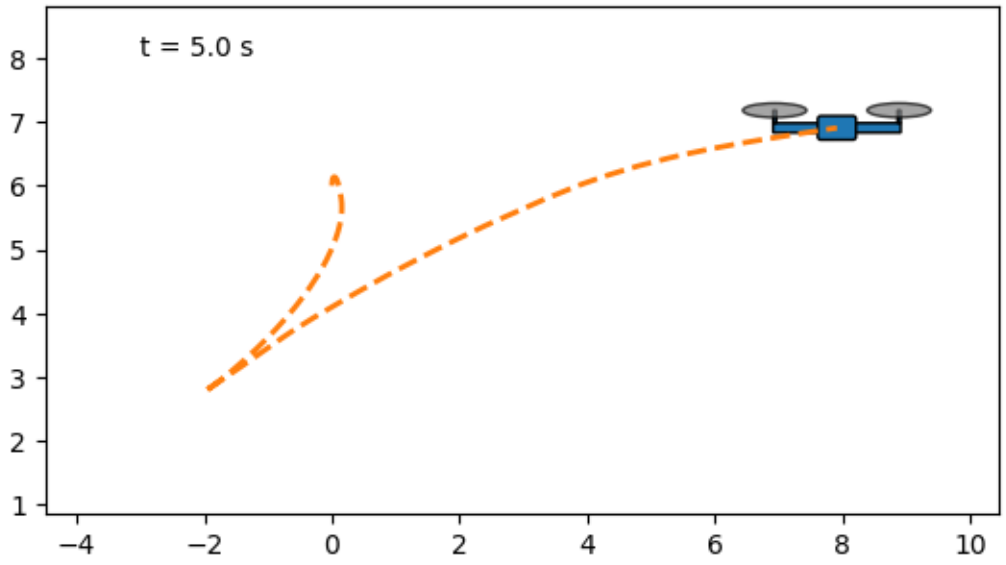
[45]:
```
# Utilities for rolling out the optimal controls and visualizing.


@jax.jit
def optimal_step(full_state, dt):
    state = full_state[2:]
    grad_value = grid.interpolate(grad_values, state)
```

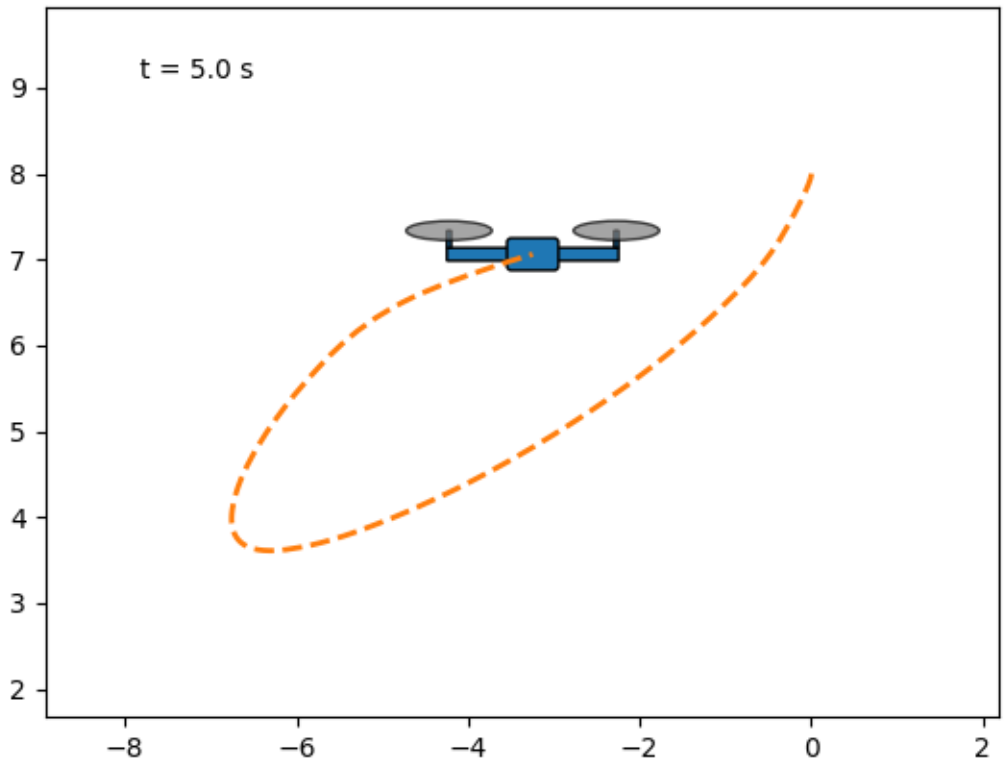
```
    control = planar_quadrotor.optimal_control(state, grad_value)
    return full_state + dt * planar_quadrotor.full_dynamics(full_state, control)


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
```

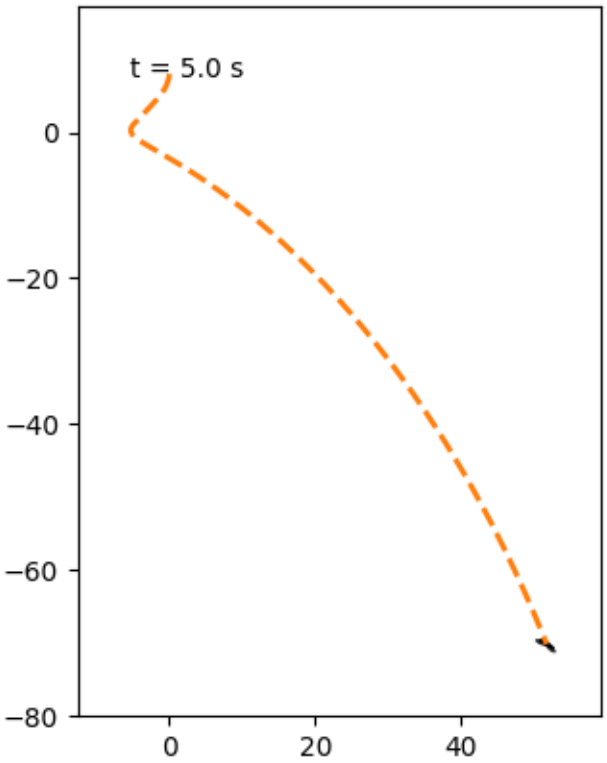
[46]:
```
# Dropping the quad straight down (v_y = -5, mimicking waiting for a sec after␣
 ↪the drop to turn the props on).
state = [5.0, -5.0, 0.0, 0.0]
fig, ani = animate_optimal_trajectory(np.array([0, 0] + state))
ani.save("planar_quad_1.mp4", writer="ffmpeg")
plt.show()

# 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()
```

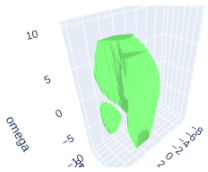


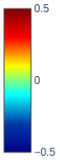



t = 5.0 s
9
8
7
6
5
4
3
2
−8
−6
−4
−2
0
2



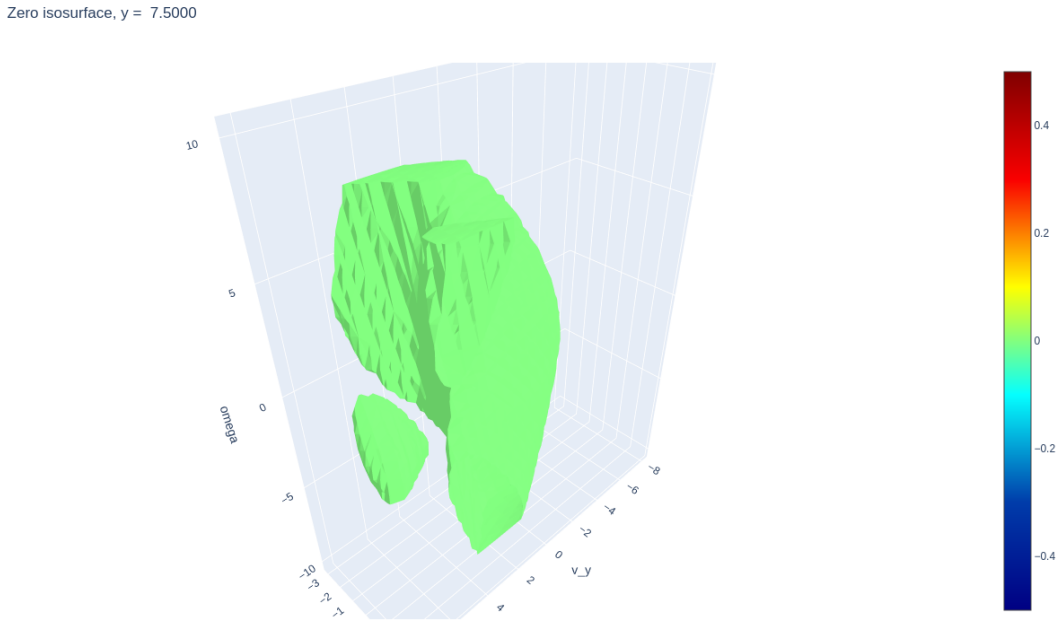
Zero isosurface, y = 7.5000

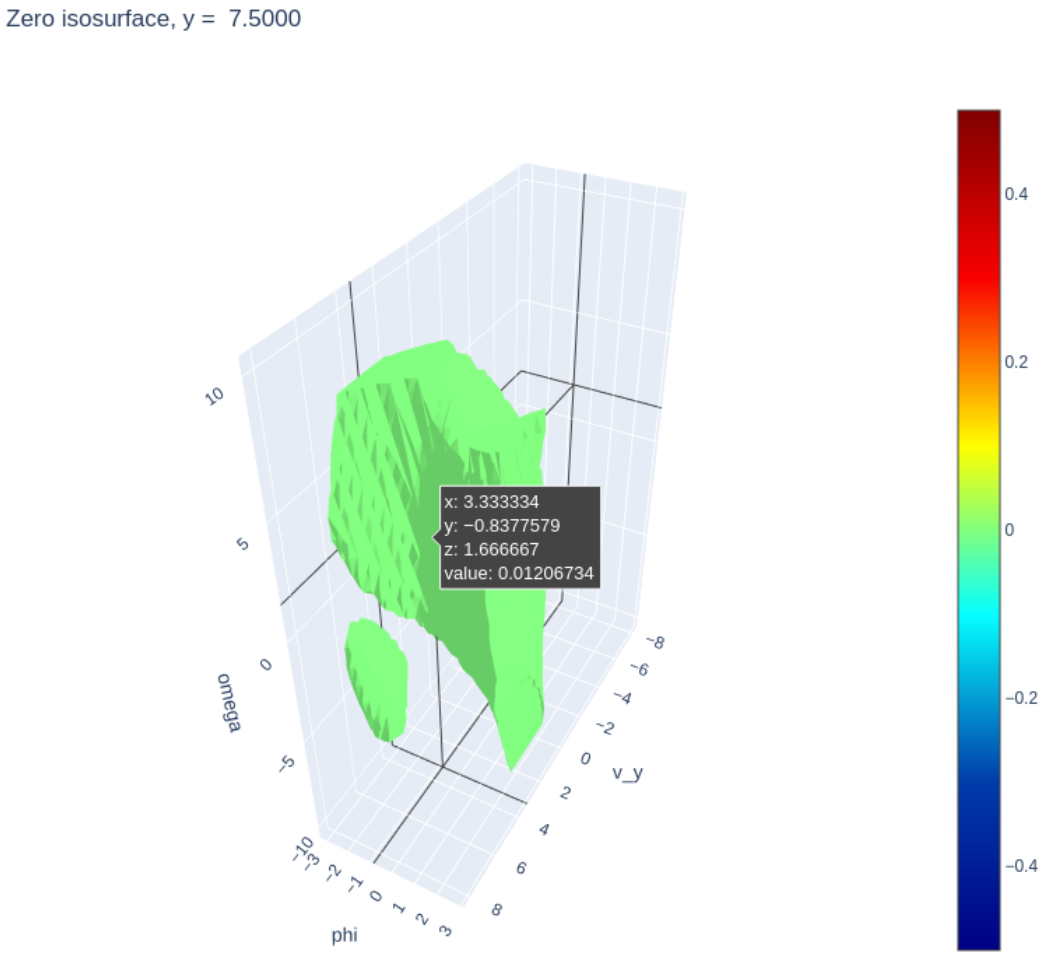


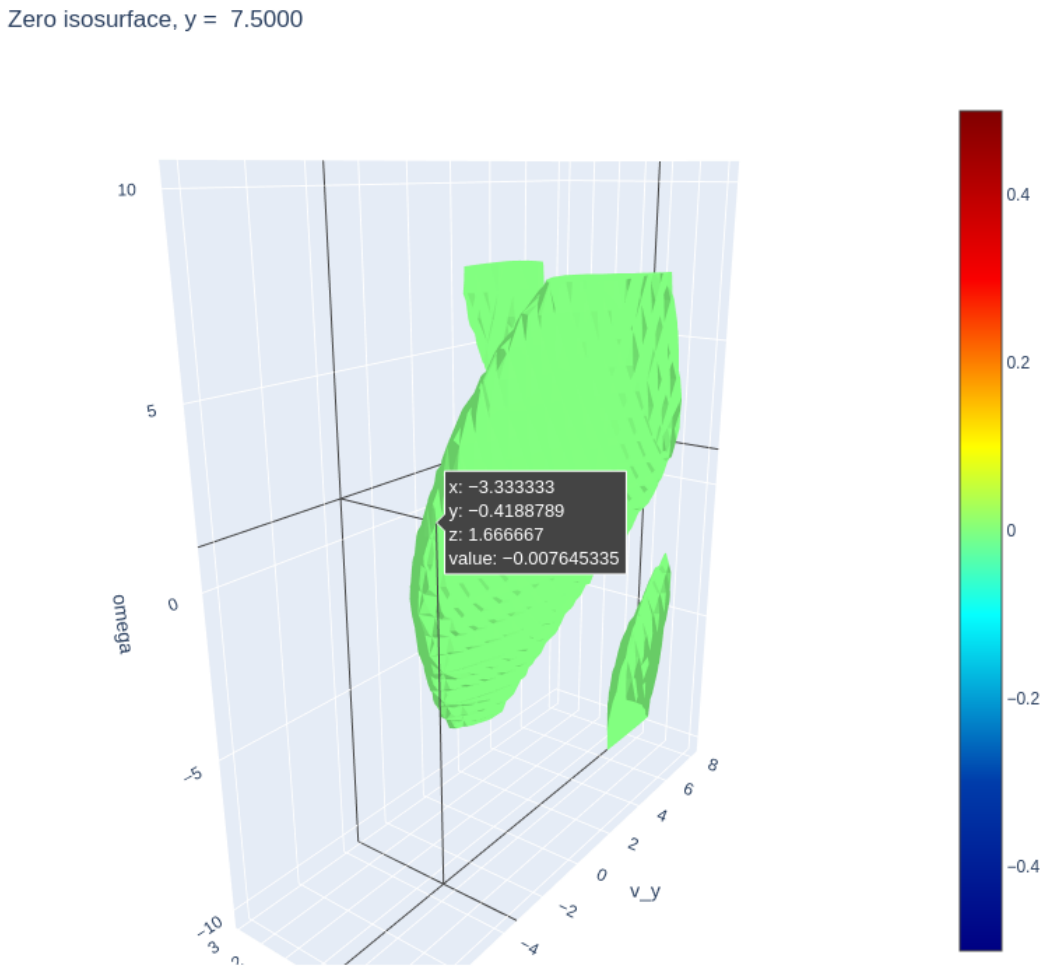


[53]:
```
# Examining an isosurface (exercise part (d)).
import plotly.graph_objects as go

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.update_layout(
    autosize=False,
    width=800,
    height=800,
)
fig.show()
```




Zero isosurface, y = 7.5000
x: 3.333334
y: −0.8377579
z: 1.666667
value: 0.01206734
omega
phi
v_y
0.4
0.2
0
−0.2
−0.4


Zero isosurface, y = 7.5000
10
5
0
-5
-10
omega
x: −3.333333
y: −0.4188789
z: 1.666667
value: −0.007645335
-4
-2
0
2
4
6
8
v_y
0.4
0.2
0
−0.2
−0.4

[ ]:

mpc_feasibility

May 19, 2024

[3]:
```
"""
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
```

0.1 Part A

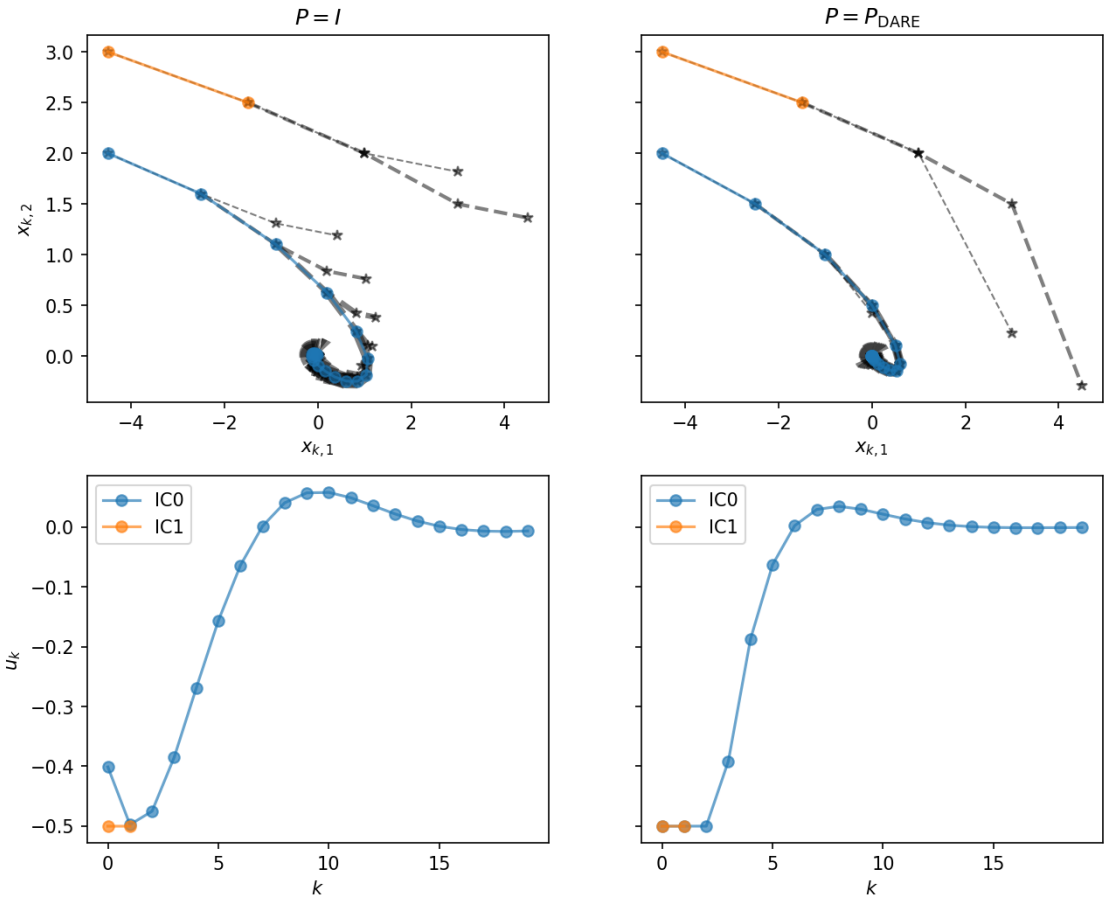
[53]:
```
def do_mpc(
    x0: np.ndarray,
    A: np.ndarray,
    B: np.ndarray,
    P: np.ndarray,
    Q: np.ndarray,
    R: np.ndarray,
    N: int,
    rx: float,
    ru: float,
    rf: float,
) -> tuple[np.ndarray, np.ndarray, str]:
    """Solve the MPC problem starting at state `x0`."""
    n, m = 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 = []

    for i in range(N):
        cost += cvx.quad_form(x_cvx[i,:], Q)
        cost += cvx.quad_form(u_cvx[i,:], R)
    cost += cvx.quad_form(x_cvx[-1,:], P)


    # state, input, and terminal constraints
    constraints += [ cvx.max(cvx.abs(x_cvx[:,:])) <= rx]
    constraints += [ cvx.max(cvx.abs(u_cvx[:-1,:])) <= ru]
    constraints += [ cvx.max(cvx.abs(x_cvx[-1,:])) <= rf]
    constraints += [ x_cvx[i+1,:] == A @ x_cvx[i,:] + B @ u_cvx[i,:] for i in␣
↪range(N)]
    constraints += [ x_cvx[0,:] == x0 ]

    # 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
```

[54]:
```
# 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 j, x0 in enumerate(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":
                print('IC', j, ' hit infeasibility')
                x_mpc = x_mpc[:t]
                u_mpc = u_mpc[:t]
                break
            x = A @ x + B @ u_mpc[t, 0, :]   # Pick out the first control out␣
 ↪of the optimized contro sequence
            ax[0, i].plot(x_mpc[t, :, 0], x_mpc[t, :, 1], "--*", color="k",␣
 ↪linewidth=1+t, alpha=1/2)
        ax[0, i].plot(x_mpc[:, 0, 0], x_mpc[:, 0, 1], "-o", alpha=2/3)
        ax[1, i].plot(u_mpc[:, 0], "-o", alpha=2/3, label='IC%d'%j)
    ax[0, i].set_title(title)
    ax[0, i].set_xlabel(r"$x_{k,1}$")
    ax[1, i].set_xlabel(r"$k$")
    ax[1, i].legend()
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()
```

```
IC 1  hit infeasibility
IC 1  hit infeasibility
```



0.1.1 Discussion Part a

Across the board, we observe that IC0 is able to converge independent of the choice of P, whereas IC1 is consistently met with infeasibility and fails.

We also observe that for IC0 which always converges, choosing P as the result of discrete algebraic Riccati equation (DARE) allows the control to reach a steady state sooner (around k=11) while the naive choice of P only approach a steady state control around k=15

0.2 Part B

[47]:
```
def compute_roa(
    A: np.ndarray,
    B: np.ndarray,
    P: np.ndarray,
    Q: np.ndarray,
    R: np.ndarray,
    N: int,                  # receding-horizon length
    rx: float,
```

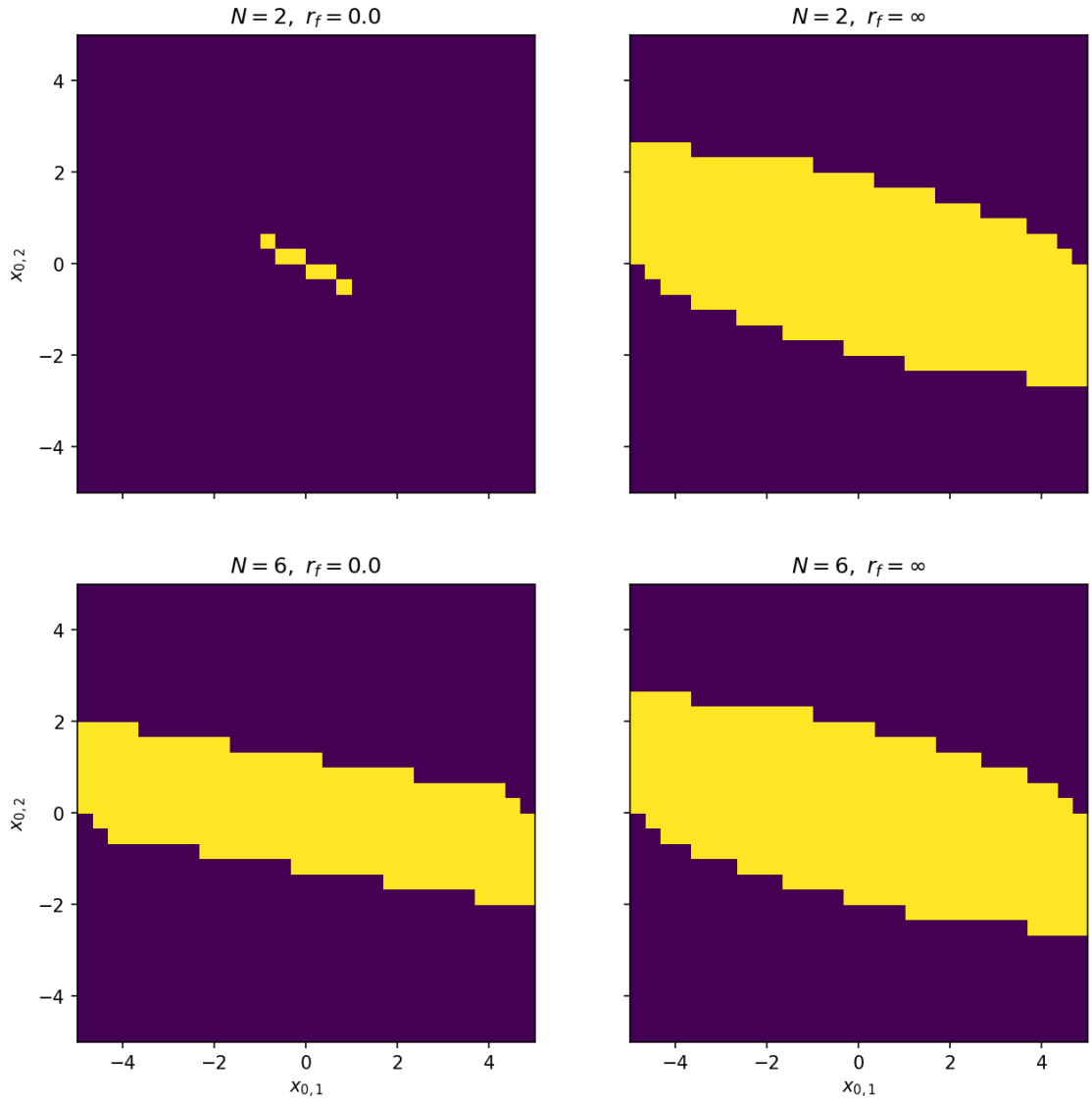
```
    ru: float,
    rf: float,
    grid_dim: int = 21,
    max_steps: int = 20,
    tol: float = 1e-2,
) -> np.ndarray:
    """Compute a region of attraction."""
    roa = np.zeros((grid_dim, grid_dim))
    xs = np.linspace(-rx, rx, grid_dim)
    for i, x1 in enumerate(xs):
        for j, x2 in enumerate(xs):
            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`.
            x_mpc = np.zeros((T, N + 1, n))
            u_mpc = np.zeros((T, N, m))
            for t in range(max_steps):
                x_mpc[t], u_mpc[t], status = do_mpc(x, A, B, P, Q, R, N, rx,␣
 ↪ru, rf)

                if status == "infeasible":
                    # print((i,x1), (j, x2), ' hit infeasibility')
                    # x_mpc = x_mpc[:t]
                    # u_mpc = u_mpc[:t]
                    roa[i,j] = 0
                    break

                x = A @ x + B @ u_mpc[t, 0, :]

                if np.max(np.abs(x)) < tol:
                    roa[i,j] = 1
                    break

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

[55]:
```
# 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()
```

```
Computing regions of attraction (this may take a while) …
```

```
  0%|          | 0/4 [00:00<?, ?it/s]
```



0.2.1 Discussion Part b

In general, we observe two trends:

- given the same terminal penalty matrix P, longer receding horizons correspond to a larger region of attraction in the state space
- given the same receding horizon length N, a less restrictive set of values that x_f can take on correspond to a larger region of attraction in the state space

[ ]:

semi_definite_program

May 19, 2024

[1]:
```
import cvxpy as cvx

import matplotlib.pyplot as plt

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)
      = np.linspace(0, 2*np.pi, num_points)
    u = np.column_stack([np.cos( ), np.sin( )])
    x = u @ L.T
    return x
```

[2]:
```
A = np.array([[0.9, 0.6],[0, 0.8]])
B = np.array([[0],[1]])
r_x = 5
r_u = 1
Q = np.eye(2)
R = np.eye(1)
P = np.eye(2)
N = 4
A, B
```

[2]:
```
(array([[0.9, 0.6],
        [0. , 0.8]]),
 array([[0],
        [1]]))
```

[3]:
```
M = cvx.Variable((2,2),symmetric=True)

constraints = []

constraints += [M<<r_x**2 * np.eye(2)]

constraints += [cvx.bmat([[M, A@M],[M@A.T, M]])>>0]

prob = cvx.Problem(
    cvx.Maximize(cvx.log_det(M)),
    constraints
)
```

[4]:
```
prob.solve()
```

[4]:
```
4.832662985874696
```

[5]:
```
M.value
```

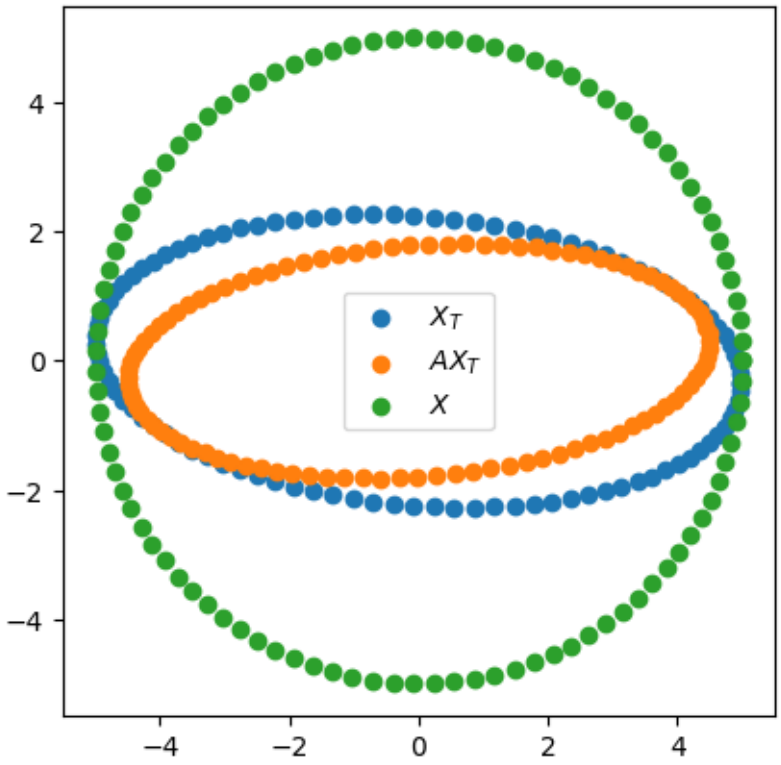
[5]:
```
array([[24.86408695, -1.64219805],
       [-1.64219805,  5.15770616]])
```

[6]:
```
np.linalg.inv(M.value)
```

[6]:
```
array([[0.04108258, 0.01308057],
       [0.01308057, 0.19804945]])
```

[7]:
```
ellipsoid_XT = generate_ellipsoid_points(M.value)
ellipsoid_A_XT = np.array([A@x for x in ellipsoid_XT])

 = np.linspace(0, 2*np.pi, 100)
ellipsoid_X = r_x*np.column_stack([np.cos( ), np.sin( )])
```

[8]:
```
fig, ax = plt.subplots()
ax.scatter(ellipsoid_XT[:,0],ellipsoid_XT[:,1], label='$X_T$')
ax.scatter(ellipsoid_A_XT[:,0],ellipsoid_A_XT[:,1], label='$AX_T$')
# ax.scatter(np.array([A@x for x in ellipsoid_A_XT])[:,0],np.array([A@x for x␣
 ↪in ellipsoid_A_XT])[:,1], label='$AAX_T$')
ax.scatter(ellipsoid_X[:,0],ellipsoid_X[:,1], label='$X$')
ax.legend()
ax.set_aspect('equal')
```



0.1 The MPC Problem

[9]:
```
x0 = np.array([[0],[-4.5]])
x0_cvx = cvx.Parameter((2,1), value=x0)

n, m = Q.shape[0], R.shape[0]

W = np.linalg.inv(M.value)

# Form the CVX Problem
cost = 0.0
constraints = []

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

# Define Cost
cost += cvx.QuadForm(x_cvx[-1,:], P)
for i in range(N):
    cost += cvx.QuadForm(x_cvx[i,:], Q)
    cost += cvx.QuadForm(u_cvx[i,:], R)
```

```
# Define Constraints
constraints += [cvx.QuadForm(x_cvx[-1,:], W)<=1]  # Contraint x.T * M^-1 * x < 1
constraints += [cvx.norm(x_cvx[i,:], 2)<=r_x for i in range(N)]
constraints += [cvx.norm(u_cvx[i,:], 2)<=r_u for i in range(N)]
constraints += [ x_cvx[i+1,:] == A @ x_cvx[i,:] + B @ u_cvx[i,:] for i in␣
 ↪range(N)]
constraints += [x_cvx[0,:]==x0_cvx[:,0]]


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

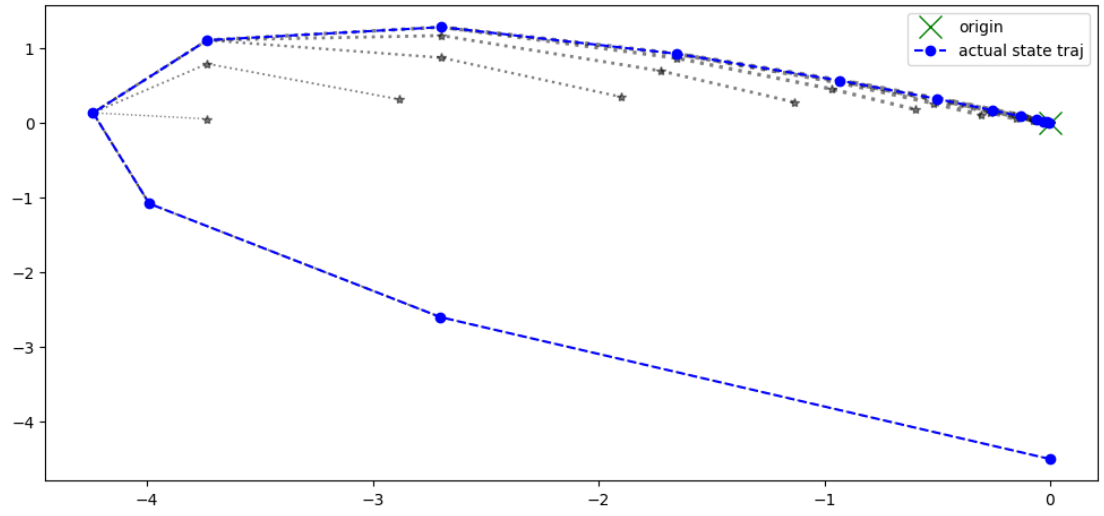
```
/home/qdeng/.pyenv/versions/3.12.1/envs/AA203/lib/python3.12/site-
packages/cvxpy/reductions/solvers/solving_chain.py:336: FutureWarning:
    Your problem is being solved with the ECOS solver by default. Starting in
    CVXPY 1.5.0, Clarabel will be used as the default solver instead. To
continue
    using ECOS, specify the ECOS solver explicitly using the ``solver=cp.ECOS``
    argument to the ``problem.solve`` method.

  warnings.warn(ECOS_DEPRECATION_MSG, FutureWarning)
```

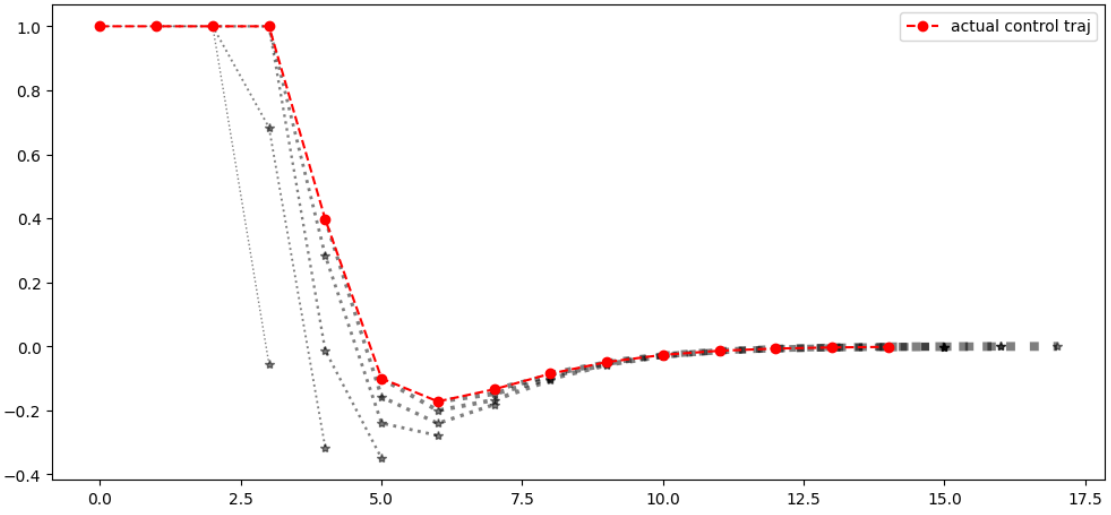
[9]:
```
86.31905795612
```

[11]:
```
T = 15
x0_cvx.value = x0
x_mpc = np.zeros((T, N + 1, n))
u_mpc = np.zeros((T, N, m))

fig, ax = plt.subplots(2,1, figsize=(12, 12))

for t in range(T):
    prob.solve()

    x_mpc[t] = x_cvx.value
    u_mpc[t] = u_cvx.value

    # set the parameter value to where the state is at the next time instance
    x0_cvx.value = A @ x0_cvx.value + B * u_mpc[t, 0, :]

    # Plot out the optimized trajectory for the horizon with length N
    ax[0].plot(x_mpc[t,:,0], x_mpc[t,:,1], 'k*:', linewidth=t*1/3+1, alpha=1/2)
    ax[1].plot(t+np.arange(0,N), u_mpc[t,:,0], 'k*:', linewidth=t*1/3+1,␣
 ↪alpha=1/2)

ax[0].plot(0,0,'gx',label='origin',markersize=15)
ax[0].plot(x_mpc[:,0,0], x_mpc[:,0,1], 'bo--', label='actual state traj')
```

```
ax[0].legend()


ax[1].plot(u_mpc[:,0,0], 'ro--', label='actual control traj')
ax[1].legend()
```

[11]: <matplotlib.legend.Legend at 0x7d8a5da37500>





[ ]: