Course: ME 550 – Nonlinear Optimal Control

Optimal Control of Multiple Drones for Obstacle Avoidance

Group #: 7

Group Members

Soheil Keshavarz

John Wang

Jiakai Wen

November 25, 2025

Table of Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Coordinating multiple aerial robots in cluttered environments is a challenging control problem due to nonlinear dynamics, tight communication constraints, and the need for real-time safety guarantees. In this project, we reproduce and study the work of Sütő et al. [1], developing a supervisory optimal control framework for multiple Parrot Mambo drones performing 3D navigation with spherical stationary obstacles and inter-drone avoidance under ideal conditions without measurement noise and communication delays.

There are two layers in the control architecture:

1. Onboard layer: a discrete-time Linear Quadratic Regulator (LQR) with a Kalman Gain for state estimation.
2. Supervisory layer: an optimization-based controller that predicts future positions and computes minimal corrections to the nominal velocity commands to ensure safety using nonsmooth barrier functions.

This approach is important because it provides a computationally efficient alternative to full Model Predictive Control (MPC), suitable for drones with limited onboard resources and uncertain network delays.

2 System Description

The paper addresses the problem of optimal control and path planning for multiple quadrotor drones operating in a three-dimensional environment with obstacle avoidance and formation constraints [1]. Controlling such systems presents several challenges. The Parrot Mambo drones modeled and simulated in the paper are shown in Figure 1. The drones exhibit nonlinear and underactuated dynamics, making stabilization and trajectory tracking nontrivial. In addition, model uncertainties, external disturbances, and unmeasurable states complicate accurate control. Path planning is also difficult, as it requires real-time re-planning when encountering unanticipated obstacles or dynamic changes in the environment. Furthermore, timing constraints imposed by limited onboard hardware restrict the complexity of algorithms that can be executed in real time, while network-induced communication delays and packet loss introduce synchronization issues and affect stability in multi-drone coordination.

To address these challenges, the paper builds upon the framework of nonsmooth barrier functions ([2]) and extends it to a more realistic three-dimensional setting with complex dynamics and communication effects. The authors propose an optimal control and planning framework that integrates obstacle avoidance through the use of nonsmooth barrier functions. Each drone operates with a baseline Linear Quadratic Regulator (LQR) controller and a Kalman filter running onboard in real time for stabilization and state estimation. On top of this, an off-board prediction-based optimization algorithm acts as a supervisory controller, determining the optimal control corrections and trajectories that minimize deviations from nominal paths while ensuring safety. This optimization is performed remotely to accommodate hardware constraints, and it explicitly compensates for network transmission delays by relying on predicted future states. The authors validated their approach through simulations using nonlinear drone models under realistic conditions with noise and delay. In this project, however, our team focused solely on an ideal

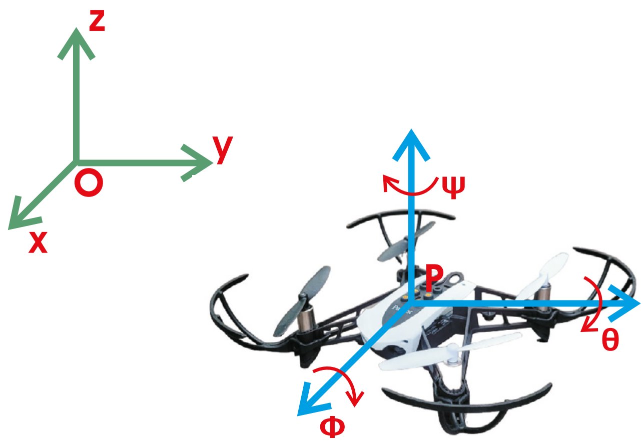

Figure 1: Parrot Mambo drone

scenario without noise or delay, given the time constraints and the scope of a course project. The overall goal is to enable multiple drones to reach their desired destinations while avoiding static and dynamic obstacles, maintaining safe distances from one another, and minimizing control effort subject to both dynamic and safety constraints.

The parameters of the Parrot Mambo drone model are shown in Table 1.

Table 1: Drone parameters

Parameter	**Notation**	**Value**	**Units**
Mass of drone	m	0.063	kg
x-axis inertia moment	I_x	5.829×10^{-4}	$\mathrm{kg \cdot m^2}$
y-axis inertia moment	I_y	7.169×10^{-4}	$\mathrm{kg \cdot m^2}$
z-axis inertia moment	I_z	1.000×10^{-3}	$\mathrm{kg \cdot m^2}$
Gravitational acceleration	g	9.8	$\frac{\mathrm{m}}{\mathrm{s}^2}$

2.1 Drone Dynamics

The original paper begins with a non-linear drone model

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}) \tag{1}$$

where the state vector

$$\mathbf{x} = \left[x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi}\right]^\top \in \mathbb{R}^n$$

contains the drone positions $\xi = [x, y, z]^\top$, Euler angles (orientation) $\eta = [\phi, \theta, \psi]^\top$, and their derivatives. The control input

$$\mathbf{u} = \left[U_{\text{coll}}, U_\phi, U_\theta, U_\psi\right]^T \in \mathbb{R}^m$$

contains thrust and body torques for roll, pitch, and yaw respectively. Refer to [1] for the non-linear model.

The model is linearized assuming small Euler angle approximations $\sin(\eta) \approx \eta$ and $\cos(\eta) = 1$ as

$$\dot{\mathbf{x}}_l(t) = \mathbf{A}\mathbf{x}_l(t) + \mathbf{B}\mathbf{u}_l(t) \tag{2}$$

which may also be written as

$$\begin{cases} \ddot{x} = \theta g \\ \ddot{y} = -\phi g \\ \ddot{z} = \dfrac{\Delta U_{\text{coll}}}{m} \end{cases} \qquad \begin{cases} \ddot{\phi} = \frac{U_\phi}{I_x} \\ \ddot{\theta} = \frac{U_\theta}{I_y} \\ \ddot{\psi} = \frac{U_\psi}{I_z} \end{cases} \tag{3}$$

where $\Delta U_{\text{coll}} = U_{\text{coll}} - mg$. The system and input matrices $\mathbf{A}$ and $\mathbf{B}$ can be written based on the definition of the state vector and model (3).

Discretizing using a first-order accurate forward difference scheme gives

$$\mathbf{x}[k+1] = \mathbf{A}_d\mathbf{x}[k] + \mathbf{B}_d\mathbf{u}[k], \tag{4}$$

with

$$\mathbf{A}_d = \mathbf{I} + T_s\mathbf{A}, \qquad \mathbf{B}_d = T_s\mathbf{B}.$$

Model (4) gives the state vector $\mathbf{x}(k+1)$ at time step $k+1$ for a sampling period of T_s. The outputs are given by

$$\mathbf{y}[k] = \mathbf{C}_d\mathbf{x}[k] \tag{5}$$

where $\mathbf{C}_d = I_{n\times n}$ measures every state.

The non-linear model is also discretized with the forward Euler scheme

$$\mathbf{x}[k+1] = \mathbf{x}[k] + T_s\mathbf{f}\big(\mathbf{x}[k], \mathbf{u}[k]\big) \tag{6}$$

for the observer design purposes.

2.2 Baseline Onboard Control

The nominal controller is an LQR state-feedback

$$\mathbf{u} = -\mathbf{K}\left(\mathbf{x} - \begin{bmatrix} r_x \\ r_y \\ r_z \\ 0_{9\times 1} \end{bmatrix}\right) \tag{7}$$

where r_x, r_y, and r_z describe the target position.

The Kalman gains are computed using model (2) with state and input weights

$$R_e = \text{diag}\big(1/15, 1000, 1000, 100\big), \qquad Q_e = \text{diag}(0.1, 0.1, 10, 0.01, 0.01, 0.01, 0.01, 0.01, 1, 0.1, 0.1, 0.1).$$

Since this is a tracking problem, the x, y, and z positions have relatively larger weights in Q_e. Additionally, the weights for z and $\dot{z}$ are much greater than their counterparts so the controller provides sufficient thrust for the drone to remain hovering. Furthermore, to reduce oscillations while the drones hover, the angular velocities have a weight 10 times larger than their respective Euler angles.

3 Supervisory Optimal Control

The continuous-time optimal control problem is formulated as

$$\begin{aligned}
\overline{\mathbf{u}}^{\text{opt}}(\overline{\mathbf{x}}) &= \arg\min_{\overline{\mathbf{u}}} \Big(\overline{\mathbf{u}}^\top \overline{\mathbf{u}} - \overline{\mathbf{u}}_{\text{nom}}^\top \overline{\mathbf{u}} \Big) \\
\text{s.t.} \quad & \nabla h_i(\overline{\mathbf{x}})^\top \overline{\mathbf{f}}(\overline{\mathbf{x}}, \overline{\mathbf{u}}) + \alpha\big(h_i(\overline{\mathbf{x}})\big) > 0, \quad i = 1, 2, \ldots, n_c.
\end{aligned}$$

The goal of the optimal control problem is to modify the control input $\overline{\mathbf{u}}$ as minimally as possible so that the drones avoid obstacles and each other. The cost function expresses how much the optimized control $\overline{\mathbf{u}}$ deviates from the nominal control input $\bar{\mathbf{u}}_{\text{nom}}$, in this case the LQR baseline controller, while encouraging less aggressive and smoother control efforts. Note that $\overline{\mathbf{x}}$ and $\overline{\mathbf{u}}$ represent the stacked dynamics of n drones. Together with m objects, the total number of constraints is given by $n_c = \frac{n(n-1)}{2} + nm$.

The constraint of the optimal control problem is a *safety constraint* known as a *control barrier function (CBF) constraint*. In essence, the constraint prescribes the minimum distance between any two drones or between a drone and an obstacle. This state-dependent inequality constraint ensures that the system's state remains within a predefined safe region $\mathcal{C} = \big\{\mathbf{x} \in \mathbb{R}^n \mid h_i(\mathbf{x}) \geq 0\big\}$, which depends on how the system evolves under the control-affine continuous-time dynamics $\dot{\mathbf{x}}(t) = f(\mathbf{x}) + G(\mathbf{x})\mathbf{u}$. Here, $h_i(\mathbf{x})$ are continuously differentiable candidate nonsmooth barrier functions corresponding to obstacle avoidance between two drones, or between a drone and an obstacle. Furthermore, $\alpha : \mathbb{R} \to \mathbb{R}$ is a locally Lipschitz, extended class-$\mathcal{K}$ function chosen to be $\alpha(h_i) = 100h_i^3$.

Consider n drones in 3D space, each with a predefined control law that drives the drones from their initial to final positions, and m ball-shaped obstacles in the environment. The goal is to modify the control input $\bar{\mathbf{u}}$ as little as possible so that the drones avoid obstacles and each other. To ensure safe flight, the minimum distance between the positions of two drones ξ_i and ξ_j should be at least r_{ij}, and the distance between the position of a drone ξ_i and an obstacle ξ_j^o should be at least r_j^o, the radii of object j. The constraints for drone-object and drone-drone interactions are defined respectively as

$$h_{ij}^o = \Big\|\xi_i - \xi_j^o\Big\|^2 - \Big(r_j^o\Big)^2 \geq 0, \qquad h_{ij}^d = \big\|\xi_i - \xi_j\big\|^2 - r_{ij}^2 \geq 0.$$

Thus in total, there are $n_c = \frac{n(n-1)}{2} + nm$ constraints which must be satisfied at timestep k.

- For drone-object interactions, the CBF constraint is equivalent to

$$2\Big(\xi_i - \xi_j^o\Big)^\top \dot{\xi}_i + 100\Big(h_{ij}^d\Big)^3 \geq 0. \tag{8}$$

- For drone-drone interactions, the CBF constraint is equivalent to

$$2\big(\xi_i - \xi_j\big)^\top \Big(\dot{\xi}_i - \dot{\xi}_j\Big) + 100\Big(h_{ij}^o\Big)^3 \geq 0. \tag{9}$$

3.1 Prediction-based Optimization

The optimal control problem discussed above is not easy to solve because the constraints involve only positions and not the full states. Additionally, implementation on the Parrot Mambo drones requires a discrete-time solution. Thus, the following optimal control problem in terms of the drone velocities $\dot{\xi}$ is proposed:

$$\begin{aligned} \dot{\xi}^{\text{opt}}[k] &= \arg\min_{\dot{\xi}} \left\| \dot{\xi}^{\text{pred}}[k] - \dot{\xi}[k] \right\|^2 \\ \text{s.t.} \quad & \nabla h_i\big(\bar{\xi}^p[k]\big)^T \dot{\xi}[k] + \alpha\Big(h_i\big(\bar{\xi}[k]\big)\Big) \geq 0, \quad i = 1, 2, \ldots, n_c. \end{aligned} \tag{10}$$

The prediction-based optimization problem takes a similar form to the previous formulation. However, the cost function now considers the Euclidean norm of the velocity difference between the optimized and nominal control. The inputs to the optimization problem are the predicted states of each drone. Additionally, the control barrier function constraint for the minimum distance between two drones or a drone and an obstacle is reformulated as a function of ξ as defined by constraints (8) and (9). The optimal control inputs are recovered from the discrete-time linear model by back-calculating it from the predicted states and the optimized velocity using finite differences.

3.2 Input Recovery

Under the linearized drone model, translational accelerations are directly controlled by the roll or pitch angles. From (3), the pitch angle, θ, produces acceleration in x direction, roll angle, ϕ, produces acceleration in y direction, and collective thrust directly controls z acceleration. Model (3) shows that the torques are directly proportional to angle accelerations.

$$\begin{aligned} &\ddot{z} = \frac{\Delta U_{\text{coll}}}{m} = \frac{U_{\text{coll}}}{m} - g \iff U_{\text{coll}} = m(\ddot{z} + g) \\ &\ddot{y} = -\phi g \iff y^{(4)} = -\ddot{\phi} g = -\frac{U_\phi}{I_x} g \iff U_\phi = -y^{(4)} \cdot \frac{I_x}{g} \\ &\ddot{x} = \theta g \iff x^{(4)} = \ddot{\theta} g = \frac{U_\theta}{I_y} g \iff U_\theta = x^{(4)} \cdot \frac{I_y}{g} \\ &\ddot{\psi} = \frac{U_\psi}{I_z} \iff U_\psi = \ddot{\psi} I_z \end{aligned}$$

Thus, the relative order of the system is 4. That is, a 4-step ahead predication is required for positions x and y, and a 2-step ahead approach is required for position z. The derivatives $\ddot{z}$, $x^{(4)}$, and $y^{(4)}$ can be numerically approximated using a finite-difference scheme of sampled drone positions. However since the optimization gives optimized velocities and not the optimized drone positions, we write an equivalent differentiation scheme in terms of future velocities velocities and the velocities at the current state. That is,

$$\begin{aligned} \ddot{f}(k) &= \frac{\dot{f}^{\text{opt}}(k+1) - \dot{f}(k)}{T_s}, \quad f = \{z\} \\ f^{(4)}(k) &= \frac{-\dot{f}(k) + 3\dot{f}^{\text{opt}}(k+1) - 3\dot{f}^{\text{opt}}(k+2) + \dot{f}^{\text{opt}}(k+3)}{T_s^3}, \quad f = \{x, y\}. \end{aligned} \tag{11}$$

Using differentiation schemes in (11), the control inputs are recovered.

$$\begin{aligned} U_{\text{coll}} &= m(\ddot{z} + g) \\ U_{\phi} &= -y^{(4)} \cdot \frac{I_x}{g} \\ U_{\theta} &= x^{(4)} \cdot \frac{I_y}{g} \\ U_{\psi} &= \ddot{\psi} I_z. \end{aligned} \tag{12}$$

We observe from (11) that the optimization problem (10) must be solved at each time step $k+1$ to $k+3$ for optimal x and y velocities and $k+1$ for z velocity. Note that the optimization problem is not solved at time step k since positions $\xi[k]$ and velocities $\dot{\xi}[k]$ of each drone are known. Thus, the optimization problem is implemented as an MPC with a horizon $H = 3$.

4 Simulation

The main simulation and control loop pseudo-code for the multi-drone control problem with collision avoidance is described in Algorithm 1. The codes are implemented in MATLAB R2023a and the problem 10 is solved using CVX [3], [4].

At a high-level, first the baseline LQR controller Kalman gains are designed offline. The main control loop continues until the Euclidean norm between drone i state and the target state is less than 0.1. Within the loop, the waypoints are first updated so the baseline controller can provide the nominal inputs for the 4-step ahead prediction of linearized model. The waypoints policy is generated along the shortest path from the drone's starting point to the goal. Waypoints are created every 30 samples and are placed at a distance of $d = 0.75\,\text{m}$ to reduce overshoot due to obstacle avoidance. This also helps with limiting the control input, hence avoiding saturation. Provided timestep $k \bmod 30 = 0$, the waypoints are updated as

$$\xi_i^{\text{ref}}[k+1] = \xi_i^{\text{ref}}[k] + d \cdot s_i \tag{13}$$

where s_i is the unit direction vector $\frac{\xi_i^{\text{target}} - \xi_i^{\text{initial}}}{\|\xi_i^{\text{target}} - \xi_i^{\text{initial}}\|}$ for drone i.

Next, collision and obstacle detection is checked between the drones and objects. If a safety constraint is violated, an active flag triggers the optimal control. The optimal control then determines the optimal velocities for future timesteps, and thus the optimal inputs at timestep k may be computed. The predicted nominal states from the LQR control are used in the optimization step. This is because there are small state perturbations and less aggressive maneuvering during drone hovering. The optimization pseudo-code is described in Algorithm 2. Given the current state $\mathbf{x}[k]$ and the optimal inputs at timestep k, the next state for the iteration loop is computed via the non-linear dynamics. Otherwise, if the active flag was not triggered, the initial state becomes the next pre-computed state $\mathbf{x}[k+1]$ from the baseline prediction.

5 Results & Discussion

The proposed control algorithm was tested using a scenario with two drones and two objects. We defined the initial positions of the robots to be $\xi_1(0) =$ (-2 0.5 0.5)$^\top$ and $\xi_2(0) =$ (-2 -0.5 -0.5)$^\top$

Algorithm 1 Main control loop with baseline LQR controller and supervisory optimal control

while $\left\|\mathbf{x}_i[k] - \mathbf{x}_i^{\text{target}}\right\| \geq 0.1$ **do**
  Waypoint update for each drone:
  if $k \bmod 30 = 0$ **then**
    for $i = 1$ to n_{drones} **do**
      $\mathbf{x}_i^{\text{ref}} \leftarrow \mathbf{x}_i^{\text{ref}} + d \cdot s_i$
      Check if terminal state reached:
      if $\left\|\mathbf{x}_i^{\text{ref}} - \mathbf{x}_i^{\text{target}}\right\| < d$ **then**
        $\mathbf{x}_i^{\text{ref}} \leftarrow \mathbf{x}_i^{\text{target}}$
      end if
    end for
  end if
  Predict 4-step ahead state trajectory using LQR:
  for $i = 1$ to n_{drones} **do**
    for $j = k$ to $k + H - 1$ **do**
      $\mathbf{u}_i[j] \leftarrow -\mathbf{K}\big(\mathbf{x}_i[j] - \mathbf{x}_i^{\text{ref}}\big)$
      $\mathbf{x}_i[j+1] \leftarrow \mathbf{A}_d \mathbf{x}_i[j] + \mathbf{B}_d \mathbf{u}_i[j]$
    end for
  end for
  Collision & obstacle detection:
  active $\leftarrow$ false
  for $i = 1$ to n_{drones} **do**
    for $j = 1$ to n_{objects} **do**
      if $\left\|\xi_i[k + H - 1] - \xi_j^{\text{obj}}\right\| \leq r_j^{\text{obj}}$ **then**
        active $\leftarrow$ true
      end if
    end for
    for $j = i + 1$ to n_{drones} **do**
      if $\big\|\xi_i[k + H - 1] - \xi_j[k + H - 1]\big\| \leq r_{ij}^{\text{drone}}$ **then**
        active $\leftarrow$ true
        break
      end if
    end for
  end for
  if *active* **then**
    Perform collision-avoidance optimization:
    $\dot{\xi}^{\text{opt}} \leftarrow \texttt{drone\_opt}(\cdot)$
    for $i = 1$ to n_{drones} **do**
      Compute numerical derivatives:
      $\ddot{z}[k] \leftarrow \frac{\dot{z}_i^{\text{opt}}[k+1] - \dot{z}_i[k]}{T_s}$
      $y^{(4)}[k] \leftarrow \frac{-\dot{y}_i[k] + 3\dot{y}_i^{\text{opt}}[k+1] - 3\dot{y}_i^{\text{opt}}[k+2] + \dot{y}_i^{\text{opt}}[k+3]}{T_s^3}$
      $x^{(4)}[k] \leftarrow \frac{-\dot{x}_i[k] + 3\dot{x}_i^{\text{opt}}[k+1] - 3\dot{x}_i^{\text{opt}}[k+2] + \dot{x}_i^{\text{opt}}[k+3]}{T_s^3}$
      $\ddot{\psi}[k] \leftarrow \frac{\dot{\psi}[k+1] - \dot{\psi}[k]}{T_s}$
      Compute optimal inputs:
      $u_{\text{coll}}^{\text{opt}}[k] \leftarrow m\big(\ddot{z}[k] + g\big)$
      $u_{\phi}^{\text{opt}}[k] \leftarrow -y^{(4)}[k] \cdot \frac{I_x}{g}$
      $u_{\theta}^{\text{opt}}[k] \leftarrow x^{(4)}[k] \cdot \frac{I_y}{g}$
      $u_{\psi}^{\text{opt}}[k] \leftarrow \ddot{\psi}[k] \cdot I_z$
      Compute updated states using non-linear dynamics:
      $\mathbf{u}_i[k] \leftarrow \big[u_{\text{coll}}, u_{\phi}, u_{\theta}, u_{\psi}\big]^T$
      $\mathbf{x}_i[k+1] \leftarrow \mathbf{x}_i[k] + T_s \mathbf{f}\big(\mathbf{x}_i[k], \mathbf{u}_i[k]\big)$
    end for
  end if
  $k \leftarrow k + 1$
end while

Algorithm 2 MPC collision-avoidance optimization, `drone_opt`

Optimization variable:
For each drone $i = 1, \ldots, n_{\text{drones}}$ and each prediction step $k = 2, \ldots, H$, $\dot{\xi}_i^{\text{opt}}[k] \in \mathbb{R}^3$.
for $i = 1$ to n_{drones} **do**
    for $k = 2$ to H **do**
        Objective function: Minimize difference between predicted and optimal velocities.
        $\text{obj} \leftarrow \text{obj} + \left\| \dot{\xi}_i^{\text{pred}}[k] - \dot{\xi}_i^{\text{opt}}[k] \right\|^2$
        Append drone-object CBF constraints:
        for $j = 1$ to n_{objects} **do**
            $2\left(\xi_i[k] - \xi_j^{\text{obj}}\right)^\top \dot{\xi}_i^{\text{opt}}[k] + 100\left(\left\|\xi_i[k] - \xi_j^{\text{obj}}\right\|^2 - \left(r_j^{\text{obj}}\right)^2\right)^3 \geq 0$
        end for
        Append drone-drone CBF constraints:
        for $j = i + 1$ to n_{drones} **do**
            $2\left(\xi_i[k] - \xi_j[k]\right)^\top \left(\dot{\xi}_i^{\text{opt}}[k] - \dot{\xi}_j^{\text{opt}}[k]\right) + 100\left(\left\|\xi_i[k] - \xi_j[k]\right\|^2 - \left(r_{ij}^{\text{drone}}\right)^2\right)^3 \geq 0$
        end for
    end for
end for
Minimize objective subject to constraints
    return $\dot{\xi}_i^{\text{opt}}[k]$ for each drone $i = 1, \ldots, n_{\text{drones}}$ and each prediction step $k = 2, \ldots, H$

and the goal positions are $\xi_{1g} = (4 \text{ -0.5 -0.5})^\top$ and $\xi_{2g} = (4\ 0.5\ 0.5)^\top$. The minimum safety distance between the two drones is 0.6 m. The obstacles' positions are $\xi_1^o = (0.3 \text{ -0.25 } 0.25)^\top$ and $\xi_1^o = (0.3\ 0.25 \text{ -0.5})^\top$, and the minimum distance from obstacles to drones is 0.3 m as in the work by Sütő et al. [1].

A sampling period of $T_s = 20$ ms is used, which balances delays and packet loss during network transmissions expected in real-deployment and capturing important transient behaviors.

5.1 Overall Trajectory Performance

Figure 2 compares trajectories generated by the constrained supervisory optimizer and the baseline LQR. The optimizer enforces inter-agent and obstacle avoidance constraints using barrier-function evaluated on predicted states. Consequently, the CVX-controlled agents deviate smoothly from the nominal straight-line path to maintain safety margins. However, the LQR drive trajectories are nearly straight and do not avoid the spherical obstacles since the controller contains no explicit safety constraints. The CVX solution therefore achieves collision-free cooperative flight, while the LQR controller would require external planning or hard safety overrides to guarantee the same behavior.

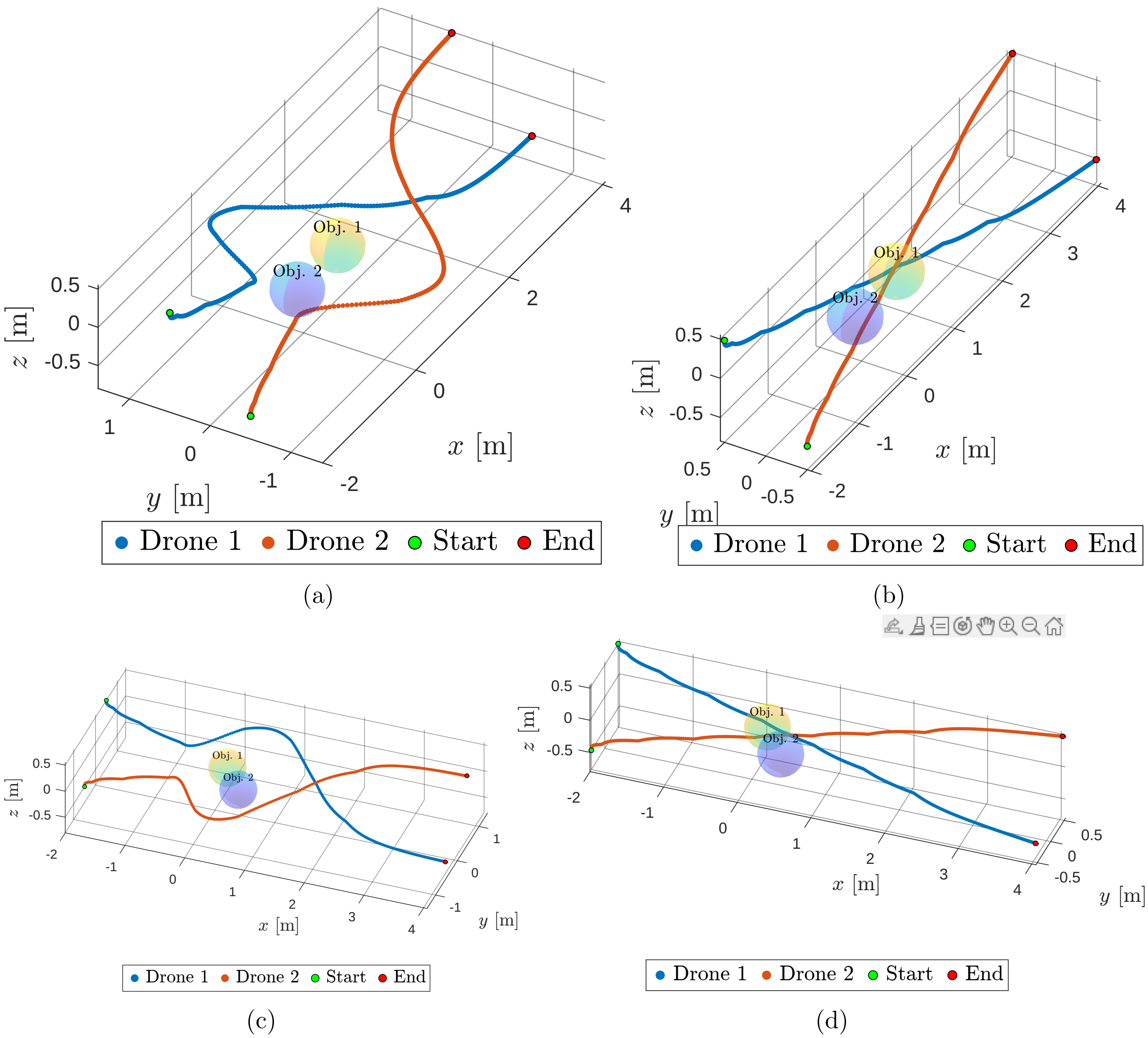

Figure 2: (a,c) The cvx solver performs the optimal control in maintaining drone-to-drone distance constraints and drone-to-obstacle distance constraints. (b,d) The baseline LQR controller outputs the path from the given initial position to the goal position without obstacle avoidance capability.

5.2 Control Inputs

As shown in Figure 3, the drone inputs of the optimal control closely match the LQR inputs except where a drone detects another drone or object. This is because the supervisory controller minimizes the deviation of the optimized velocities from the LQR velocities. Since the inputs in (12) are a function of velocity, the deviation of optimal inputs from nominal inputs are similarly minimized. Additionally, U_{coll} stabilizes at a constant value of $mg = 0.6174\,\text{N}$, which is the minimum thrust required for the drones to hover constantly at their target positions. Furthermore, a repeating step-wise behavior is exhibited in the inputs U_{coll} and U_θ. This is because the reference of the LQR control is regulated by waypoints every 30 samples. The behaviour is exhibited in U_{coll} (thrust) and U_θ (pitch) because both inputs are related to the elevation climb of the drones.

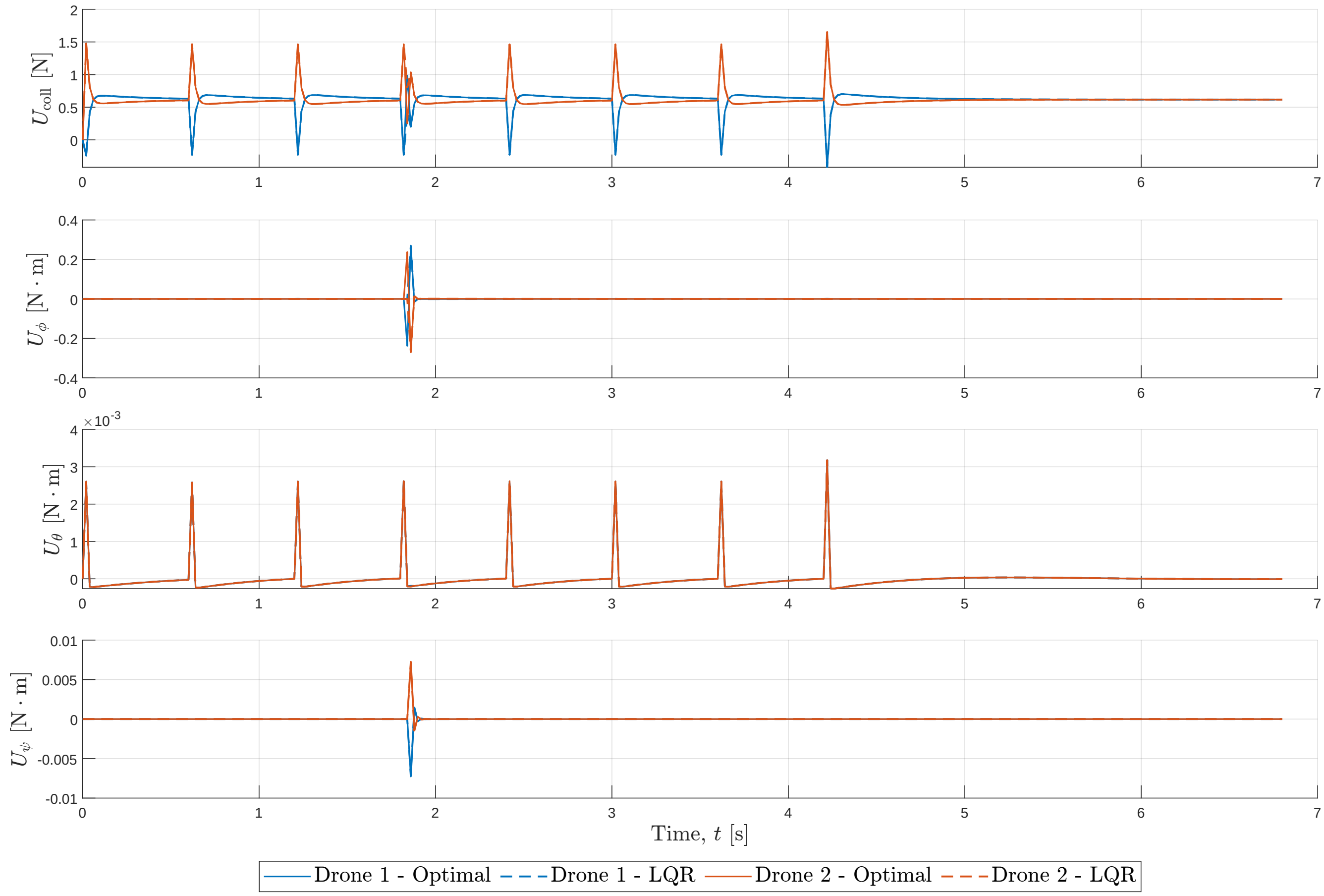

Figure 3: Control inputs applied to the drones.

5.3 Drone-Drone Distance Results

Figure 4 plots the inter-drone distance over the simulation. The red dashed line indicates the required minimum separation of 0.6 m. The controller maintains a comfortable separation for most of the trial, settling near 1.3 m in steady state. However, two brief near constraint violations are visible (around 4 s and 7 s), where the inter-drone distance drops closer to 0.6 m. These transients indicate that the supervisory optimizer prevents momentary undercutting to meet the safety margin. After the transients the controller recovers, returning the pair to a safe separation.

5.4 Drone-Object Distance Results

Figure 5 shows the distances from each drone to Object 1 (top) and Object 2 (bottom), with the red dashed line indicating the required minimum stand-off distance of 0.3 m. Both drones begin several meters away from each obstacle at their pre-defined initial positions and approach them as they progress toward the goal. As the trajectories bring the agents closer to the obstacle boundaries, the control barrier function constraints become active, causing a smooth deflection in each path. Importantly, neither drone violates the minimum allowed distance at any point in the simulation; the closest approach occurs around 4–6 s, with a minimum distance of approximately 0.7–1.0 m. Drone 1 consistently comes slightly closer to the objects than Drone 2, reflecting coordinated avoidance where each agent adjusts its path relative to both the environment and

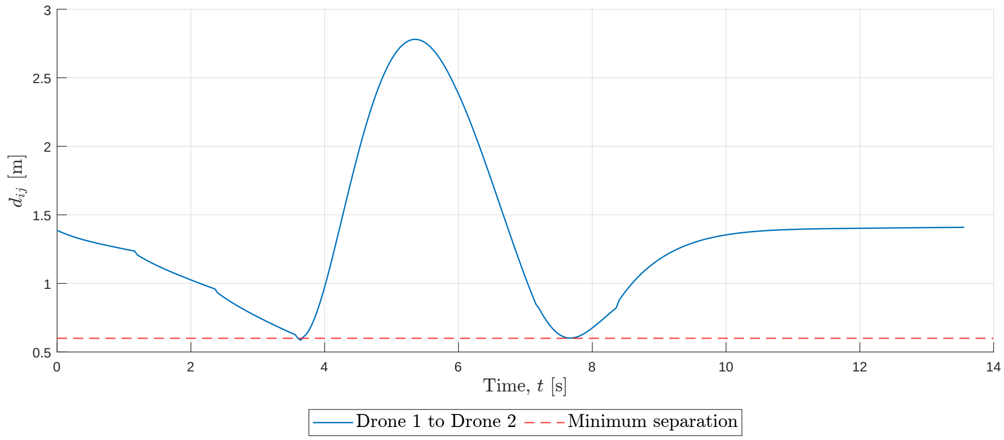

Figure 4: Drone to Drone distance constraint satisfied.

the other drone. After passing the obstacles, the distances increase monotonically as the drones move toward the goal position. These results demonstrate that the CVX MPC controller enforces obstacle-avoidance constraints reliably and proactively, maintaining smooth and safe separation throughout the maneuver.

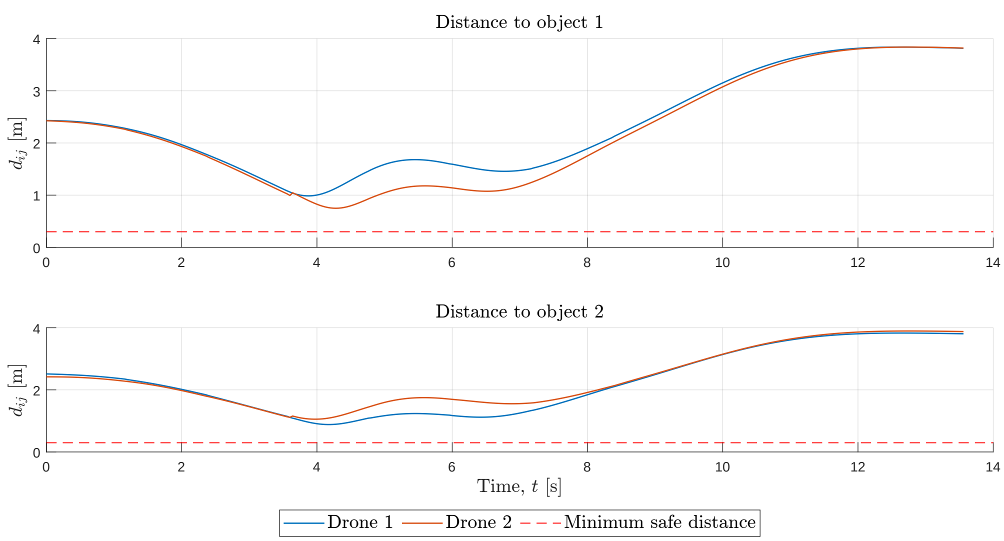

Figure 5: Distance constraints simulation for two drones and two obstacles during flight.

5.5 State Transition Results

Overall, drone 1 demonstrates a well-coordinated obstacle avoidance trajectory, shown by smooth forward progression and vertical maneuvering. As shown in Figure 6, the simulation shows the drone 1 is able to start from the given initial positions and reach the final given states. It maintains steady forward motion in the x direction, traveling from start to end position in about 6s with a relatively constant velocity that peaks around 5s. The lateral motion in the y direction reveals significant maneuvering activity. It is accompanied by substantial velocity variations reaching 2 m/s near 2s, indicating active avoidance behavior. The z-direction motion showcases continuous altitude adjustments to maintain safe separation from both obstacles and drone 2. From roll, ϕ,

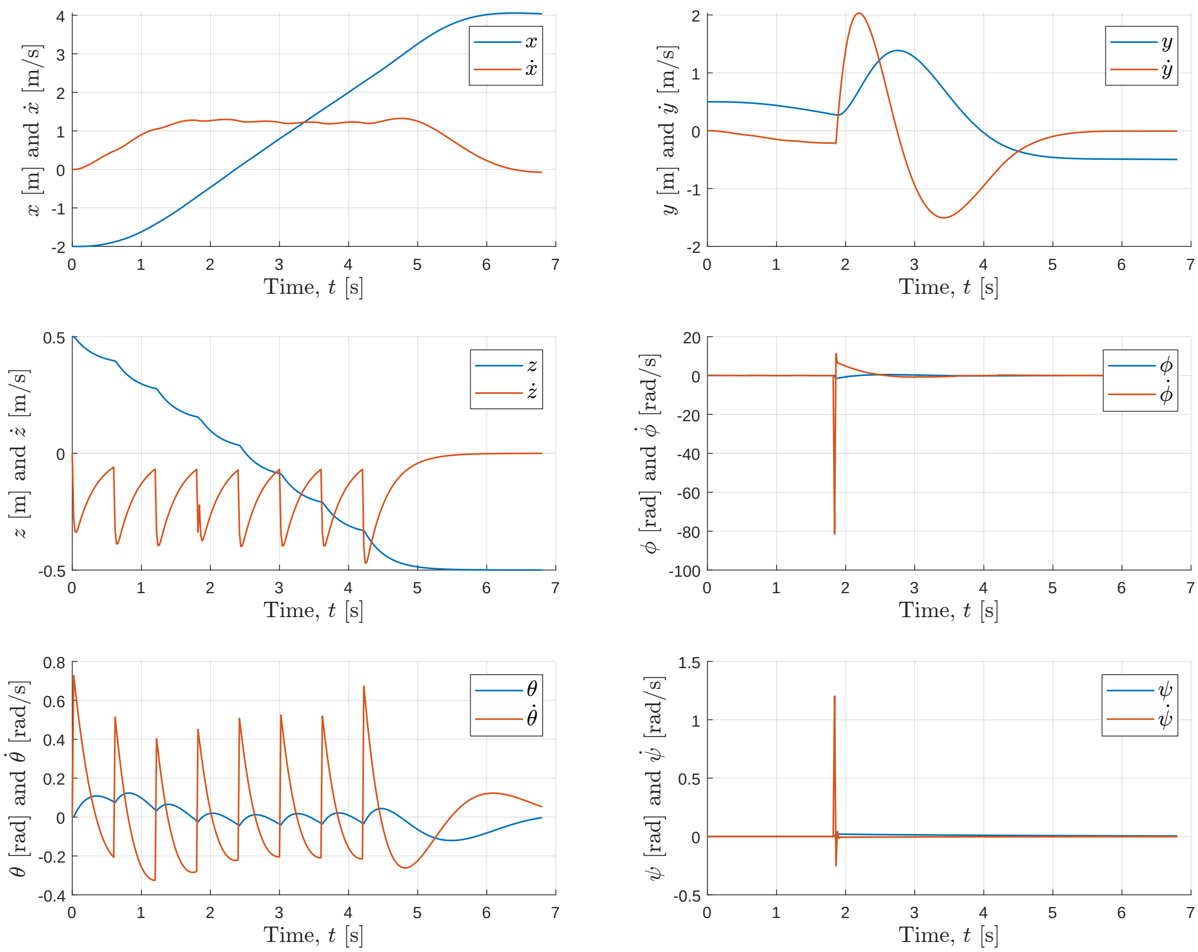

Figure 6: All the input state positions and velocities for drone 1 during flight.

and yaw, ψ, plots in Figure 6, the most aggressive input occur around 2s. This timing corresponds precisely to the critical obstacle encounter phase visible in the 3D trajectory plot. By 6.8s, all Euler angles converge to near-zero values, and all velocities approach zero, demonstrating successful completion of the avoidance maneuver. The smooth nature of all position trajectories, despite the aggressive control inputs, indicates that the CVX solver successfully generated feasible solutions that respect the system's dynamic constraints while satisfying both obstacle avoidance and inter-drone separation requirements.

Drone 2 exhibits complementary behavior to drone 1, as shown in Figure 7, executing a coordinated avoidance strategy that maintains safe separation while navigating the same obstacle environment. Similar to drone 1, drone 2 maintains smooth forward progression in the x-direction with comparable velocity profiles, ensuring both drones make forward progress without collision. The y-direction velocity shows large oscillations similar in magnitude to drone 1 but phase-shifted, peaking around 3s with values of approximately 1.5 m/s. The most distinctive characteristic of drone 2's trajectory is its ascending vertical maneuver, which directly contrasts with drone 1's descending strategy. The control effort patterns mirror those of drone 1, with peak roll control inputs occurring around 2s and reaching similar magnitudes of 80 rad/s^2. The pitch angle exhibit high-frequency oscillations that gradually dampen, indicating active stabilization throughout the maneuver, while the yaw plot shows a brief but sharp adjustment around 2s. By 6.8s, all of drone 2's states have converged to the final values with zero velocities and near-zero attitude angles, con-

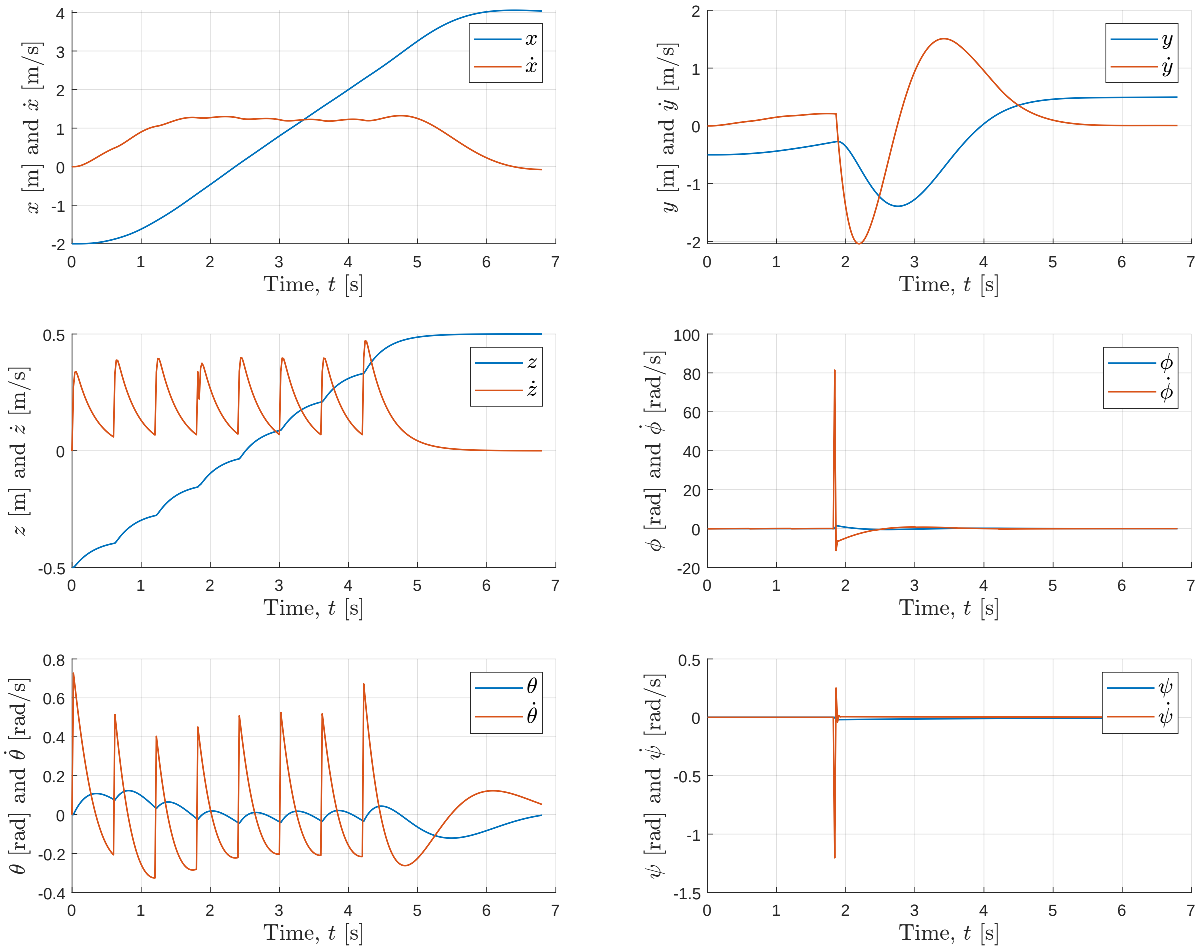

Figure 7: All the input state positions and velocities for drone 2 during flight.

firming successful mission completion. The complementary nature of drone 2's maneuvers relative to drone 1 demonstrates the effectiveness of the CVX-based trajectory optimization in generating coordinated multi-drone solutions that simultaneously satisfy obstacle avoidance constraints, inter-drone distance constraints, and dynamic feasibility requirements.

6 Conclusion

A baseline controller was designed using LQR with state-feedback to transition the drones from their initial position to final position via waypoints. Since the LQR controller does not provide obstacle or drone avoidance, a supervisory controller was implemented using optimal control. If a drone detected another drone or object within a prescribed tolerance, optimization was triggered. The optimization minimizes the deviation of the optimal drone velocities from the predicted velocities using LQR subject to the constraints of the control barrier function for safety. Once the optimized velocities were determined, the optimal inputs at the current time step could be determined using finite differences at different sampled values. The optimal inputs were then fed into the non-linear model and used in the simulation. Using the supervisory controller, the results showed that the drones were able to navigate safely from their initial positions to the target positions without colliding with other drones or obstacles. Furthermore, the optimal inputs deviated minimally from the nominal inputs based on the optimization problem.

Our team encountered several challenges throughout this project. The first difficulty involved understanding the control constraint, specifically the control barrier function, and reconstructing the sequence of steps leading to the framework presented in Algorithm 1. The original paper provided these steps in a scattered manner, which required additional effort to interpret. A more significant challenge was the input-recovery process, which was only briefly described in the paper. To address this, our team reviewed and discussed the derivation of the control input from the optimized velocities multiple times to gain clarity. Finally, we faced implementation challenges related to the optimization in Algorithm 2. Although the original work indicates running the optimization once per prediction step, our team opted instead to perform a full MPC optimization that computes the optimal velocities for the entire prediction horizon within a single optimization run. Further work could be performed to account for process noise and delay which is inherent in drone systems.

References

[1] B. Sütő, A. Codrean, and Z. Lendek, "Optimal control of multiple drones for obstacle avoidance," *IFAC-PapersOnLine*, vol. 56, no. 2, pp. 5475–5481, 2023, 22nd IFAC World Congress, ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2023.10.200. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405896323005517.

[2] P. Glotfelter, J. Cortés, and M. Egerstedt, "Nonsmooth barrier functions with applications to multi-robot systems," *IEEE Control Systems Letters*, vol. 1, no. 2, pp. 310–315, 2017. DOI: 10.1109/LCSYS.2017.2710943.

[3] I. CVX Research, *CVX: Matlab software for disciplined convex programming, version 2.0*, https://cvxr.com/cvx, Aug. 2012.

[4] M. Grant and S. Boyd, "Graph implementations for nonsmooth convex programs," in *Recent Advances in Learning and Control*, ser. Lecture Notes in Control and Information Sciences, V. Blondel, S. Boyd, and H. Kimura, Eds., http://stanford.edu/~boyd/graph_dcp.html, Springer-Verlag Limited, 2008, pp. 95–110.

A Supplementary drone state plots

The states of each drone for the simulation scenario are shown in Figures 8 and 9.

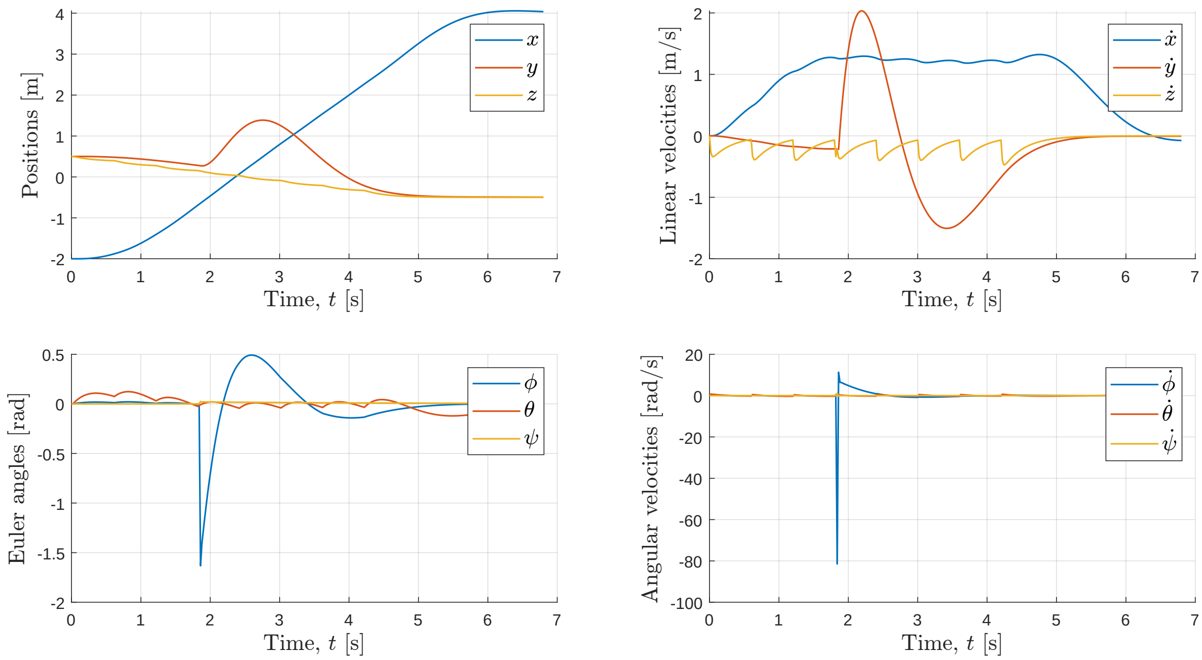

Figure 8: All the input state positions and velocities for drone 1 during flight.

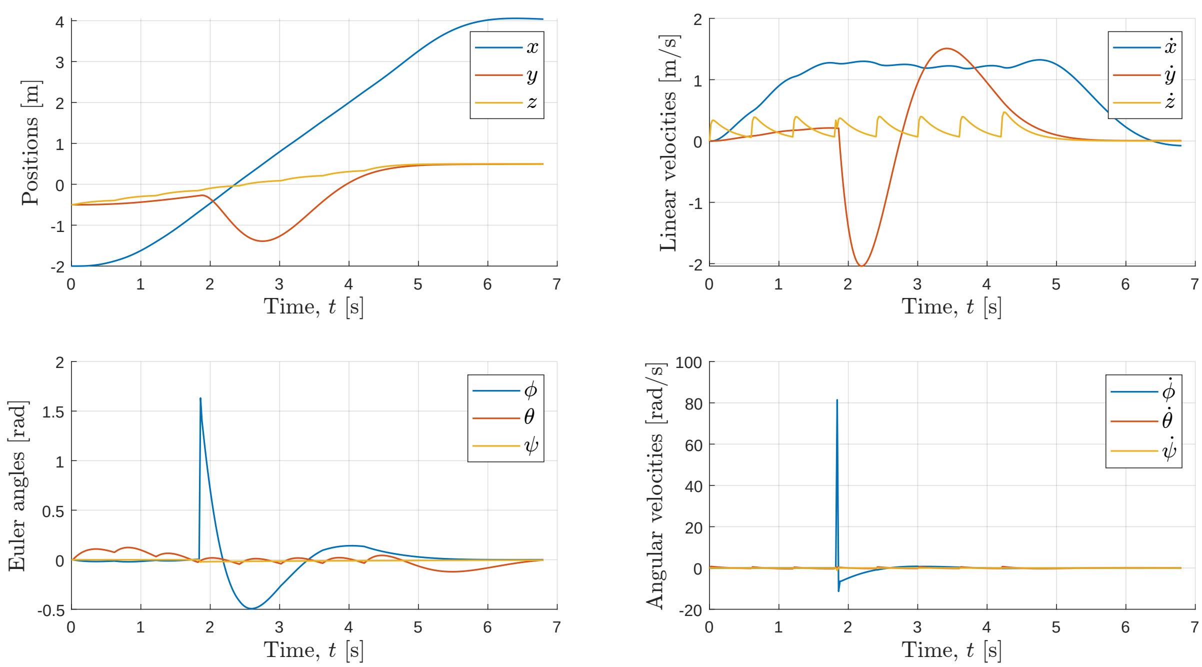

Figure 9: All the input state positions and velocities for drone 2 during flight.

B MATLAB code

Main control loop and simulation

main

```
clear; close all; clc;

%% Optimization approach
opts = ["cvx_mpc", "lqr"];
opt  = "cvx_mpc";

%% Sampling rate
Ts = 0.02;

%% Linearized drone dynamics
[A, B, Ad, Bd] = helper.linearized_drone(Ts);

%% Non-linear drone dynamics
f = str2func("helper.f");

%% Dimensions
[n, m] = size(B); % States (dim n), inputs (dim m)

%% LQR Design

% Cost matrices
R_e = diag([1/15; 1000; 1000; 100]);
Q_e = diag([0.1; 0.1; 10; 0.01; 0.01; 0.01; 0.01; 0.01; 1; 0.1; 0.1; 0.1]);

% LQR gains
[K, ~, ~] = lqrd(A, B, Q_e, R_e, Ts);

%% Scenario Setup

% Simulation variables
H = 5;          % prediction horizon
n_drones  = 2; % number of drones
n_objects = 2; % number of obstacles

% Drone parameters
m_drone = 0.063; % kg
I_x = 0.5829e-4; % kg m^2
I_y = 0.7169e-4; % kg m^2
I_z = 1.000e-4;  % kg m^2
g = 9.8;          % m/s^2

% Initialize states and control inputs
X = zeros(n, n_drones, H);
```

```
U = zeros(m, n_drones, H);

% Initial position for each drone (x0, y0, z0)
X(:, 1, 1) = [-2; 0.5; 0.5; zeros(n - 3, 1)];
X(:, 2, 1) = [-2; -0.5; -0.5; zeros(n - 3, 1)];

% Target position for each drone (xf, yf, zf)
X_target = zeros(n, n_drones);
X_target(1:3, 1) = [4; -0.5; -0.5];
X_target(1:3, 2) = [4; 0.5; 0.5];

% Safety radii
r_safe = 0.0; % Design parameter
r_drone = 0.6;

% Obstacle positions and radii
P_objects = [[0.3; -0.25; 0.25], [0.3; 0.25; -0.5]]; % Array of column vectors (x, y,
    z)
R_objects = [0.3, 0.3];                                % Radii of objects

% Unit vector for shortest path of each drone
s = zeros(n, n_drones);

% Reference position
X_ref = zeros(n, n_drones);

% Setup
d = 0.75;                  % Waypoint distance
for i = 1:n_drones
    p0 = X(1:3, i, 1);     % Initial position of drone i
    pf = X_target(1:3, i); % Target position of drone i
    v  = pf - p0;          % Direction vector
    s(1:3, i) = v/norm(v); % Normalized direction

    X_ref(1:3, i) = p0;
end

% Data log
x_log = cell(n_drones, 1);
u_log = cell(n_drones, 1);
u_lqr_log = cell(n_drones, 1);
d_obj_log = cell(n_drones, n_objects);
d_drone_log = cell(n_drones, n_drones);

for i = 1:n_drones
    x_log{i} = X(:, i, 1);
    u_log{i} = U(:, i, 1);
    u_lqr_log{i} = U(:, i, 1);
end

```

```
% Helper class
util = helper(r_drone, r_safe, P_objects, R_objects, n_drones, n_objects, Ts, opt);

%% Simulation variable
iter = 0;                          % Iteration count
waypoint = true(n_drones, 1);    % Waypoint terminal flag
terminated = false(n_drones,1); % Target position achieved flag

%% Main loop
while any(~terminated)
    % Stop condition
    if iter > 500
        break;
    end

    % Check per-drone termination
    for i = 1:n_drones
        if ~terminated(i)
            if vecnorm(X(:, i, 1) - X_target(:, i), 2, 1) < 0.1
                terminated(i) = true;
                disp(['Iteration ', num2str(iter), ': Drone ', num2str(i), ' has terminated.']);
            end
        end
    end

    % Compute waypoint
    if mod(iter, 30) == 0
        % Update waypoint
        X_ref(:, waypoint) = X_ref(:, waypoint) + d*s(:, waypoint);

        % Compute distance to goal
        e = X_ref - X_target;
        dist = vecnorm(e, 2, 1);

        % Set reference if at goal
        terminal = dist < d;
        X_ref(:, terminal) = X_target(:, terminal);

        % Update waypoint flag
        waypoint = ~terminal;
    end

    %% Predict linearized state-dynamics for each drone using baseline controller (LQR)
    for i = 1:n_drones
        if terminated(i), continue; end

        for j = 1:(H-1)
            % Basline control input via LQR
```

```
            U(:, i, j) = -K*(X(:, i, j) - X_ref(:, i));
            X(:, i, j + 1) = Ad*X(:, i, j) + Bd*U(:, i, j);
            U(1, i, j) = U(1, i, j) + m_drone*g; % For completeness
        end

        u_lqr_log{i} = [u_lqr_log{i}, U(:, i, 1)];
    end

    %% Optimization trigger flag
    active = false;

    %% Collision and obstacle detection
    for i = 1:n_drones
        if terminated(i), continue; end

        % Drone i position (x, y, z)
        P_i = X(1:3, i, end);

        % Drone-to-object check
        for j = 1:n_objects
            P_j = P_objects(:, j);
            d_ij = norm(P_i - P_j);
            d_obj_log{i, j} = [d_obj_log{i, j}, d_ij];

            if d_ij <= (R_objects(j) + r_safe)
                active = true;
                disp(['Drone ', num2str(i), ' near obstacle ', num2str(j), '. Constraint violated.']);
                break;
            end
        end
        if active, break; end

        % Drone-to-drone check
        for j = (i+1):n_drones
            if terminated(j), continue; end
            P_j = X(1:3, j, end);
            d_ij = norm(P_i - P_j);
            d_drone_log{i, j} = [d_drone_log{i, j}, d_ij];

            if d_ij <= r_drone
                active = true;
                disp(['Drone ', num2str(i), ' near drone ', num2str(j), '. Constraint violated.']);
                break;
            end
        end
        if active, break; end
    end

```

```
    %% Apply optimization when needed
    if active && opt ~= "lqr"
        disp(['Iteration: ', num2str(iter), ' - Optimization active']);

        % Optimization
        k = 1;
        dP_sol = util.drone_opt(X, k + 3); % 3-step ahead optimization

        for i = 1:n_drones
            if terminated(i), continue; end

            % Apply finite differences and linearized dynamics to find U_coll (related to z)
            zdot = X(9, i, :);
            zddot = (dP_sol(3, i, 1) - zdot(1))/Ts;

            % Apply finite differences and linearized dynamics to find U_phi (related to y)
            ydot = X(8, i, :);
            yddot = (-ydot(1) + 3*dP_sol(2, i, 1) - 3*dP_sol(2, i, 2) + dP_sol(2, i, 3))/Ts^3;

            % Apply finite differences and linearized dynamics to find U_theta (related to x)
            xdot = X(7, i, :);
            xddot = (-xdot(1) + 3*dP_sol(1, i, 1) - 3*dP_sol(1, i, 2) + dP_sol(1, i, 3))/Ts^3;

            % Apply finite differences and linearized dynamics to find U_psi (related to psi)
            psidot = X(12, i, :);
            psiddot = (psidot(2) - psidot(1))/Ts;

            % Optimal control inputs
            U_coll = (zddot + g)*m_drone;
            U_phi = yddot*(-I_x/g);
            U_theta = xddot*(I_y/g);
            U_psi = psiddot*(I_z);

            %% Compute non-linear dynamics for optimal control
            u0 = [U_coll; U_phi; U_theta; U_psi];
            x0 = X(:, i, 1);
            x0 = x0 + Ts*f(x0, u0);

            %% Update variables and log trajectory
            X(:, i, 1) = x0;
            x_log{i} = [x_log{i}, x0];
            u_log{i} = [u_log{i}, u0];
        end
    else
```

```
        disp(['Iteration: ', num2str(iter), ' - LQR active']);

        for i = 1:n_drones
            if terminated(i), continue; end

            %% Compute linearized dynamics for LQR
            x0 = X(:, i, 2);
            u0 = U(:, i, 1);

            %% Update variables and log trajectory
            X(:, i, 1) = x0;
            x_log{i} = [x_log{i}, x0];
            u_log{i} = [u_log{i}, u0];
        end
    end

    %% Collision and obstacle detection post-check
    for i = 1:n_drones
        if terminated(i), continue; end

        % Drone i position (x, y, z)
        P_i = X(1:3, i, 1);

        % Drone-to-object check
        for j = 1:n_objects
            P_j = P_objects(:, j);
            d_ij = norm(P_i - P_j);
            d_obj_log{i, j} = [d_obj_log{i, j}, d_ij];

            if d_ij <= (R_objects(j) + r_safe)
                disp(['Drone ', num2str(i), ' near obstacle ', num2str(j), '. Constraint violated.']);
            end
        end

        % Drone-to-drone check
        for j = (i+1):n_drones
            if terminated(j), continue; end
            P_j = X(1:3, j, 1);
            d_ij = norm(P_i - P_j);
            d_drone_log{i, j} = [d_drone_log{i, j}, d_ij];

            if d_ij <= r_drone
                disp(['Drone ', num2str(i), ' near drone ', num2str(j), '. Constraint violated.']);
            end
        end
    end

    %% Increment iteration
```

```
    iter = iter + 1;
end

%% Plot results
util.plotTrajectories(x_log, opt);

do_plot = false;
if do_plot && opt ~= "lqr"
    util.plotControlInputs(u_log, u_lqr_log, opt);
    util.plotStateTransitions(x_log, opt);
    util.plotGroupedStates(x_log, opt);
    util.plotDroneObjectDistances(d_obj_log, opt);
    util.plotDroneDroneDistances(d_drone_log, opt);
end

do_animate = false;
if do_animate
    util.animateDrones(x_log, opt);
end
```

Helper class with drone model, optimization, and plotting

helper.m

```
classdef helper
    properties
        r_drone
        r_safe
        P_objects
        R_objects
        n_drones
        n_objects
        Ts
        opt
    end

    %% ===========================================================
    % Constructor
    methods
        function self = helper(r_drone, r_safe, P_objects, R_objects, n_drones,
    n_objects, Ts, opt)
            self.r_drone   = r_drone;
            self.r_safe    = r_safe;
            self.P_objects = P_objects;
            self.R_objects = R_objects;
            self.n_drones  = n_drones;
            self.n_objects = n_objects;
            self.Ts        = Ts;
```

```
            self.opt       = opt;
        end
    end

    methods (Static)
        %% ==========================================================
        % Linearized dynamics
        function [A, B, Ad, Bd] = linearized_drone(Ts)
            % Ts is the sampling period, this function outputs continunous A, B if Ts
            % is empty.
            % Discrete Ad and Bd using forward Euler
            % state x is 12x1
            % input u is 4x1

            % this allows input with or without Ts
            if nargin < 1
                Ts = [];
            end

            % system parameters
            m  = 0.063; % kg
            I_x = 0.5829e-4; % kg*m^2
            I_y = 0.7169e-4; % kg*m^2
            I_z = 1.000e-4; % kg*m^2
            g  = 9.8; % m/s^2

            % Continuous time A, B
            A = zeros(12,12);

            A(1,7) = 1;
            A(2,8) = 1;
            A(3,9) = 1;
            A(4,10)= 1;
            A(5,11)= 1;
            A(6,12)= 1;

            % simplified model
            A(7,5) = g; % x_2dot
            A(8,4) = -g; % y_2dot

            B = zeros(12,4);

            % u_l = u - u_e
            B(9,1)  = 1/m; % this only computes U_coll/m, full term should be z_2dot =
     U_coll/m - g
            B(10,2) = 1/I_x;
            B(11,3) = 1/I_y;
            B(12,4) = 1/I_z;

            % if we have Ts, discretized time via forward Euler:
```

```
        if ~isempty(Ts)
            Ad = eye(12) + Ts * A;
            Bd = Ts * B;
        else
            Ad = []; Bd = [];
        end
    end

    %% ===========================================================
    % Non-linear dynamics
    function xdot = f(x, u)
        % continous time dynamics
        % state x is 12x1
        % input u is 4x1

        % system parameters
        m = 0.063; % kg
        I_x = 0.5829e-4; % kgm^2
        I_y = 0.7169e-4; % kgm^2
        I_z = 1.000e-4; % kgm^2
        g = 9.8; % m/s^2

        % state
        p_x = x(1);
        p_y = x(2);
        p_z = x(3);
        phi = x(4);
        theta = x(5);
        psi = x(6);
        v_x = x(7);
        v_y = x(8);
        v_z = x(9);
        phi_dot = x(10);
        theta_dot = x(11);
        psi_dot = x(12);

        % input
        U_coll = u(1);
        U_phi  = u(2);
        U_theta= u(3);
        U_psi  = u(4);

        % representation trigs
        cphi = cos(phi);
        sphi = sin(phi);
        ctheta = cos(theta);
        stheta = sin(theta);
        cpsi = cos(psi);
        spsi = sin(psi);

```

```
        % Jacobian matrix (phi, theta, psi)
        J11 = I_x; J12 = 0; J13 = -I_x*stheta;
        J21 = 0; J22 = I_y*cphi^2 + I_z*sphi^2; J23 = (I_y - I_z)*cphi*sphi*ctheta;
        J31 = -I_x*stheta; J32 = (I_y - I_z)*cphi*sphi*ctheta;
        J33 = I_x*stheta^2 + I_y*sphi^2*ctheta^2 + I_z*cphi^2*ctheta^2;
        J = [J11, J12, J13;
             J21, J22, J23;
             J31, J32, J33];
        % J = diag([I_x, I_y, I_z]);

        % Coriolis matrix (phi, theta, psi, phi_dot, theta_dot, psi_dot)
        c11 = 0;
        c12 = (I_y - I_z)*(theta_dot*cphi*sphi + psi_dot*sphi^2*ctheta) + ...
              (I_z - I_y)*psi_dot*cphi^2*ctheta - I_x*psi_dot*ctheta;
        c13 = (I_z - I_y)*psi_dot*(ctheta^2)*sphi*cphi;

        c21 = (I_z - I_y)*(theta_dot*sphi*cphi + psi_dot*sphi^2*ctheta) + ...
              (I_y - I_z)*psi_dot*cphi^2*ctheta + I_x*psi_dot*ctheta;
        c22 = (I_z - I_y)*phi_dot*cphi*sphi;
        c23 = -I_x*psi_dot*stheta*ctheta + I_y*psi_dot*sphi^2*stheta*ctheta + ...
              I_z*psi_dot*cphi^2*stheta*ctheta;

        c31 = (I_y - I_z)*psi_dot*(ctheta^2)*sphi*cphi - I_x*theta_dot*ctheta;
        c32 = (I_z - I_y)*(theta_dot*cphi*sphi*stheta + phi_dot*sphi^2*ctheta) + ...
              (I_y - I_z)*phi_dot*cphi^2*ctheta + I_x*psi_dot*stheta*ctheta - ...
              I_y*psi_dot*sphi^2*stheta*ctheta - I_z*psi_dot*cphi^2*stheta*ctheta;
        c33 = (I_y - I_z)*phi_dot*(ctheta^2)*sphi*cphi - ...
              I_y*theta_dot*sphi^2*stheta*ctheta - I_z*theta_dot*cphi^2*stheta*ctheta + ...
              I_x*theta_dot*stheta*ctheta;

        c = [c11, c12, c13;
             c21, c22, c23;
             c31, c32, c33];

        x2dot = (cphi*stheta*cpsi + sphi*spsi) * U_coll / m;
        y2dot = (cphi*stheta*cpsi - sphi*spsi) * U_coll / m;
        z2dot = -g + (cphi*ctheta) * U_coll / m;

        U = [U_phi; U_theta; U_psi];
        eta_dot = [phi_dot; theta_dot; psi_dot];

        paren = U - c*eta_dot;
        eta2dot = J \ (U - c*eta_dot);

        phi2dot = eta2dot(1);
        theta2dot = eta2dot(2);
        psi2dot = eta2dot(3);
```

```matlab

            % xdot
            xdot = zeros(12,1);

            xdot(1) = v_x;
            xdot(2) = v_y;
            xdot(3) = v_z;
            xdot(4) = phi_dot;
            xdot(5) = theta_dot;
            xdot(6) = psi_dot;

            xdot(7)  = x2dot;
            xdot(8)  = y2dot;
            xdot(9)  = z2dot;
            xdot(10) = phi2dot;
            xdot(11) = theta2dot;
            xdot(12) = psi2dot;
        end
    end

    methods
        %% ============================================================
        % Drone optimization, MPC
        function dP_sol = drone_opt(self, X, H)
            cvx_begin quiet
                % Initialize variables
                variables dP(3, self.n_drones, H - 1)
                obj = 0;
                constraints = [];

                % Optimize from k = 2 to H
                for i = 1:self.n_drones
                    for k = 2:H
                        %% Compute objective of drones
                        dv = X(7:9, i, k) - dP(:, i, k - 1);
                        obj = obj + dv'*dv;

                        % Drone i position at timestep k
                        P_i = X(1:3, i, k);

                        %% Drone-to-object constraints
                        for j = 1:self.n_objects
                            % Object j position
                            P_j = self.P_objects(:, j);

                            % Control barrier function
                            d_ij = P_i - P_j;
                            h_ij = d_ij'*d_ij - (self.R_objects(j) + self.r_safe)^2;
                            grad_h = 2*d_ij;
                            alpha = 100*h_ij^3;
```

```
                        constraints = [constraints, grad_h'*dP(:, i, k - 1) + alpha >= 0];
                    end

                    %% Drone-to-drone constraints
                    for j = (i+1):self.n_drones
                        % Drone j position at timestep k
                        P_j = X(1:3, j, k);

                        % Control barrier function
                        d_ij = P_i - P_j;
                        h_ij = d_ij'*d_ij - self.r_drone^2;
                        grad_h = 2*[d_ij; -d_ij];
                        dP_stack = [dP(:, i, k - 1); dP(:, j, k - 1)];
                        alpha = 100*h_ij^3;
                        constraints = [constraints, grad_h'*dP_stack + alpha >= 0];
                    end
                end
            end

            minimize(obj)
            subject to
                constraints;
        cvx_end

        % Store optimized solution
        dP_sol = dP;
    end

    %% ===========================================================
    % 3D trajectories
    function plotTrajectories(self, x_log, opt)
        % Camera views
        views = [[-60,35]; [20,35]];
        colors = lines(self.n_drones);

        for v = 1:size(views,1)
            % Figure setup
            fig = figure();

            hold on; grid on;
            view(views(v,1), views(v,2));
            xlabel('$x$ [m]','Interpreter','latex');
            ylabel('$y$ [m]','Interpreter','latex');
            zlabel('$z$ [m]','Interpreter','latex');
            axis equal;

            ax = gca;
            ax.SortMethod = 'childorder';
```

```
        % Label sizes
        ax.XLabel.FontSize = 14;
        ax.YLabel.FontSize = 14;
        ax.ZLabel.FontSize = 14;
        ax.GridAlpha = 0.5;

        % Plot drone trajectories
        h_drones  = gobjects(self.n_drones,1);
        start_pts = zeros(3,self.n_drones);
        end_pts   = zeros(3,self.n_drones);

        for i = 1:self.n_drones
            xd = x_log{i};
            start_pts(:,i) = xd(1:3,1);
            end_pts(:,i)   = xd(1:3,end);

            h_drones(i) = scatter3( ...
                xd(1,:), xd(2,:), xd(3,:), ...
                5, colors(i,:), 'filled');
        end

        % Plot obstacles
        for j = 1:self.n_objects
            [Xs, Ys, Zs] = sphere(20);

            Xs = self.R_objects(j)*Xs + self.P_objects(1,j);
            Ys = self.R_objects(j)*Ys + self.P_objects(2,j);
            Zs = self.R_objects(j)*Zs + self.P_objects(3,j);

            s = surf(Xs, Ys, Zs, Zs);
            s.EdgeColor = 'k';
            s.FaceColor = 'interp';
            s.FaceAlpha = 0.45;

            text( ...
                self.P_objects(1,j), self.P_objects(2,j), ...
                self.P_objects(3,j) + 0.75*self.R_objects(j), ...
                sprintf('Obj. %d', j), ...
                'Interpreter','latex', ...
                'HorizontalAlignment','center', ...
                'FontSize',8, 'FontWeight','bold');
        end

        shading interp
        colormap(parula)

        % Plot start/end markers
        h_start = gobjects(self.n_drones,1);
        h_end   = gobjects(self.n_drones,1);
```

```

            for i = 1:self.n_drones
                h_start(i) = scatter3( ...
                    start_pts(1,i), start_pts(2,i), start_pts(3,i), ...
                    10, 'g', 'filled', 'o', 'MarkerEdgeColor','k');

                h_end(i) = scatter3( ...
                    end_pts(1,i), end_pts(2,i), end_pts(3,i), ...
                    10, 'r', 'filled', 'o', 'MarkerEdgeColor','k');
            end

            % Legend
            legend( ...
                [h_drones; h_start(1); h_end(1)], ...
                [ arrayfun(@(d) sprintf('Drone %d',d), 1:self.n_drones, 'UniformOutput', false), ...
                  {'Start'}, {'End'} ], ...
                'Interpreter','latex', ...
                'Location','southoutside', ...
                'Orientation','horizontal', 'FontSize', 14);

            % Export figure
            exportgraphics(fig, ...
                "figures/3D_trajectory_" + opt + "_view" + v + ".pdf", ...
                'ContentType','vector');
        end
    end

    %% ============================================================
    % Control inputs (4x1 tiledlayout)
    function plotControlInputs(self, u_log, u_lqr_log, opt)
        % Control input labels and units
        u_labels = {'$U_{\mathrm{coll}}$', '$U_{\phi}$', '$U_{\theta}$', '$U_{\psi}$'};
        u_units  = {'$\mathrm{N}$', '$\mathrm{N}\cdot\mathrm{m}$', ...
                    '$\mathrm{N}\cdot\mathrm{m}$', '$\mathrm{N}\cdot\mathrm{m}$'};

        colors = lines(self.n_drones);
        m = 4;

        % Create figure
        fig = figure('Visible','off','Position',[100 100 1200 800]);
        t = tiledlayout(4,1,'TileSpacing','compact','Padding','compact');

        % Build legend text
        legend_entries = cell(1, 2*self.n_drones);
        for i = 1:self.n_drones
            legend_entries{2*i-1} = sprintf('Drone %d - Optimal', i);
            legend_entries{2*i}   = sprintf('Drone %d - LQR', i);
        end
```

```matlab

            % Plot control inputs
            for u_row = 1:m
                ax = nexttile;
                hold(ax,'on');
                grid(ax,'on');

                for i = 1:self.n_drones
                    time_u   = (0:size(u_log{i},2)-1)   * self.Ts;
                    time_lqr = (0:size(u_lqr_log{i},2)-1) * self.Ts;

                    plot(ax, time_u,   u_log{i}(u_row,:),     'LineWidth',1,   'Color',colors(i,:));
                    plot(ax, time_lqr, u_lqr_log{i}(u_row,:), '--', 'LineWidth', 1.2, 'Color',colors(i,:));
                end

                ylabel(ax, u_labels{u_row} + " [" + u_units{u_row} + "]", 'Interpreter','latex','FontSize',14);
                if u_row == m
                    xlabel(ax, 'Time, $t$ [s]', 'Interpreter','latex','FontSize',14);
                end
            end

            % Add legend
            legend(legend_entries, 'Interpreter','latex', ...
                'Location','southoutside', 'Orientation','horizontal', 'FontSize', 14);

            % Export figure
            exportgraphics(fig, "figures/control_inputs_" + opt + ".pdf");
        end

        %% ============================================================
        % State vs state-derivative (per drone)
        function plotStateTransitions(self, x_log, opt)
            % State labels for positions and velocities
            state_labels = {'$x$','$y$','$z$','$\phi$','$\theta$','$\psi$', ...
                            '$\dot{x}$','$\dot{y}$','$\dot{z}$', ...
                            '$\dot{\phi}$','$\dot{\theta}$','$\dot{\psi}$'};
            state_units = {'[m]','[m]','[m]','[rad]','[rad]','[rad]', ...
                           '[m/s]','[m/s]','[m/s]', ...
                           '[rad/s]','[rad/s]','[rad/s]'};

            for i = 1:self.n_drones
                % Extract data
                xdata = x_log{i};
                T = size(xdata,2);
                time = (0:T-1) * self.Ts;

```

```
            % Create figure
            fig = figure('Visible','off','Position',[100 100 1200 900]);

            % Plot 6 position and angle states with their derivatives
            for k = 1:6
                subplot(3,2,k);
                hold on;
                grid on;

                plot(time, xdata(k,:),   'LineWidth',1);
                plot(time, xdata(k+6,:), 'LineWidth',1);

                xlabel('Time, $t$ [s]', 'Interpreter','latex','FontSize',14);
                label = state_labels(k) + " " + state_units(k) + " and " +
    state_labels(k+6) + " " + state_units(k+6);
                ylabel(label,  'Interpreter','latex','FontSize',14);

                legend({state_labels{k}, state_labels{k+6}}, 'Interpreter','latex'
    , 'FontSize', 14);
            end

            % Export figure
            exportgraphics(fig, ...
                "figures/drone" + i + "_state_transitions_" + opt + ".pdf");
        end
    end

    %% ============================================================
    % Grouped states overview (2x2 per drone)
    function plotGroupedStates(self, x_log, opt)
        % State groups: indices, labels, titles
        triples = {
            [1 2 3],      {'$x$','$y$','$z$'},                             '
    Positions [m]';
            [7 8 9],      {'$\dot{x}$','$\dot{y}$','$\dot{z}$'},           '
    Linear velocities [m/s]';
            [4 5 6],      {'$\phi$','$\theta$','$\psi$'},                  '
    Euler angles [rad]';
            [10 11 12],   {'$\dot{\phi}$','$\dot{\theta}$','$\dot{\psi}$'},   '
    Angular velocities [rad/s]'
        };

        for i = 1:self.n_drones
            % Extract data and time vector
            xdata = x_log{i};
            time = (0:size(xdata,2)-1) * self.Ts;

            % Create figure
            fig = figure('Visible','off','Position',[100 100 1200 600]);

```

```
                % Plot four groups of states
                for sp = 1:4
                    idxs   = triples{sp,1};
                    labels = triples{sp,2};
                    title_label = triples{sp,3};

                    subplot(2,2,sp);
                    hold on;
                    grid on;

                    for k = 1:3
                        plot(time, xdata(idxs(k),:), 'LineWidth',1);
                    end

                    ylabel(title_label, 'Interpreter','latex','FontSize',14);
                    xlabel('Time, $t$ [s]', 'Interpreter','latex','FontSize',14);
                    legend(labels, 'Interpreter','latex', 'FontSize', 14);
                end

                % Export figure
                exportgraphics(fig, ...
                    "figures/drone" + i + "_states_" + opt + ".pdf");
            end
        end

        %% ============================================================
        % Drone-object distances
        function plotDroneObjectDistances(self, d_obj_log, opt)
            % Colors for each drone
            colors = lines(self.n_drones);

            % Create figure sized by number of objects
            fig = figure('Visible','off', 'Position',[100 100 1200 300*self.n_objects]);

            for j = 1:self.n_objects
                subplot(self.n_objects,1,j);
                hold on;
                grid on;

                leg = {};

                % Plot distance from each drone to object j
                for i = 1:self.n_drones
                    dvec = d_obj_log{i,j};
                    time = (0:length(dvec)-1) * self.Ts;

                    % Plot every second point to reduce size
                    plot(time(1:2:end), dvec(1:2:end), 'LineWidth',1, 'Color', colors(i,:));
```

```
                    leg{end+1} = sprintf('Drone %d', i);
                end

                % Minimum safe distance line
                yline(self.r_safe + self.R_objects(j), 'r--', 'LineWidth',1);
                leg{end+1} = 'Minimum safe distance';

                % Label sizes
                ax.FontSize = 10;
                ax.XLabel.FontSize = 12;
                ax.YLabel.FontSize = 12;

                if j == self.n_objects
                    xlabel('Time, $t$ [s]', 'Interpreter','latex','FontSize',14);
                end

                ylabel('$d_{ij}$ [m]', 'Interpreter','latex','FontSize',14);
                title(sprintf('Distance to object %d', j), 'Interpreter','latex','FontSize',14);
            end

            % Combined legend at bottom
            legend(leg, 'Interpreter','latex', ...
                'Location','southoutside', 'Orientation','horizontal', 'FontSize',14);

            % Export figure
            exportgraphics(fig, "figures/drone_obj_dist_" + opt + ".pdf");
        end

        %% ============================================================
        % Drone-drone distances
        function plotDroneDroneDistances(self, d_drone_log, opt)
            % Colors for each drone
            colors = lines(self.n_drones);

            % Create figure
            fig = figure('Visible','off','Position',[100 100 1200 500]);

            hold on;
            grid on;

            leg = {};

            % Plot pairwise drone distances
            for i = 1:self.n_drones
                for j = i+1:self.n_drones
                    dvec = d_drone_log{i,j};
                    time = (0:length(dvec)-1) * self.Ts;

                    % Downsample to reduce plot size
```

```
                plot(time(1:2:end), dvec(1:2:end), ...
                    'LineWidth',1, 'Color', colors(i,:));

                leg{end+1} = sprintf('Drone %d to Drone %d', i, j);
            end
        end

        % Minimum allowed distance line
        yline(self.r_drone, 'r--', 'LineWidth',1);
        leg{end+1} = 'Minimum separation';

        xlabel('Time, $t$ [s]', 'Interpreter','latex','FontSize',14);
        ylabel('$d_{ij}$ [m]', 'Interpreter','latex','FontSize',14);

        legend(leg, 'Interpreter','latex', ...
            'Location','southoutside', 'Orientation','horizontal','FontSize',14);

        exportgraphics(fig, "figures/drone_drone_dist_" + opt + ".pdf");
    end

    %% ===========================================================
    % Multi-drone animation
    function animateDrones(self, x_log, opt)
        gif_name = "3D_trajectory_" + opt + ".gif";
        fps = 20;
        delay = 1/fps;

        % Compute fixed axis limits
        all_x = []; all_y = []; all_z = [];
        for i = 1:self.n_drones
            traj = x_log{i};
            all_x = [all_x traj(1,:)];
            all_y = [all_y traj(2,:)];
            all_z = [all_z traj(3,:)];
        end

        margin = 0.2;
        xmin = min(all_x) - margin;
        xmax = max(all_x) + margin;
        ymin = min(all_y) - margin;
        ymax = max(all_y) + margin;
        zmin = min(all_z) - margin;
        zmax = max(all_z) + margin;

        % Setup figure
        fig_anim = figure('Visible','off', 'Position',[100 100 1600 1200], '
    Renderer', 'opengl');

        hold on; grid on;
        view(20,35);
```

```
        xlabel('$x$ [m]','Interpreter','latex');
        ylabel('$y$ [m]','Interpreter','latex');
        zlabel('$z$ [m]','Interpreter','latex');

        axis equal;
        xlim([xmin xmax]);
        ylim([ymin ymax]);
        zlim([zmin zmax]);

        ax = gca;
        ax.SortMethod = 'childorder';

        % Label sizes
        ax.FontSize = 10;
        ax.XLabel.FontSize = 12;
        ax.YLabel.FontSize = 12;
        ax.ZLabel.FontSize = 12;

        colors = lines(self.n_drones);

        % Drone geometry model (simplified quadrotor, ~0.18 x 0.18 x 0.10 m)

        % Body box dimensions
        bx = 0.10;   % length (x)
        by = 0.03;   % width  (y)
        bz = 0.03;   % height (z)
        dx = bx/2; dy = by/2; dz = bz/2;

        % Body vertices (centered at origin)
        bodyVerts = [ ...
            -dx -dy -dz;
             dx -dy -dz;
             dx  dy -dz;
            -dx  dy -dz;
            -dx -dy  dz;
             dx -dy  dz;
             dx  dy  dz;
            -dx  dy  dz];

        bodyFaces = [ ...
            1 2 3 4;
            5 6 7 8;
            1 2 6 5;
            2 3 7 6;
            3 4 8 7;
            4 1 5 8];

        % Arms (4 arms at +/- 45 degrees, square cross-section)
        arm_half = 0.09;      % arm length from center to near rotor
```

```
        arm_w    = 0.01;      % arm thickness
        a        = arm_w/2;

        % Base arm along +x
        baseArm = [ ...
            0        -a  -a;
            arm_half -a  -a;
            arm_half  a  -a;
            0         a  -a;
            0        -a   a;
            arm_half -a   a;
            arm_half  a   a;
            0         a   a];

        Rz_deg = @(ang) [cosd(ang) -sind(ang) 0; ...
                          sind(ang)  cosd(ang) 0; ...
                          0          0         1];

        armVerts = cell(1,4);
        armFaces = repmat({[1 2 3 4; 5 6 7 8; 1 2 6 5; 2 3 7 6; 3 4 8 7; 4 1 5
8]}, 1, 4);
        armAngles = [45, 135, -135, -45];   % front-right, front-left, rear-left,
rear-right

        for a_i = 1:4
            armVerts{a_i} = (Rz_deg(armAngles(a_i)) * baseArm')';
        end

        % Motors (small cylinders) and their offsets from origin
        N_cyl = 20;
        r_motor = 0.012;
        h_motor = 0.02;

        [Xc, Yc, Zc] = cylinder(r_motor, N_cyl);
        Zc = Zc * h_motor;

        motorX = Xc;
        motorY = Yc;
        motorZ = Zc;

        motorOffsets = zeros(4,3);
        motorOffsets(1,:) = (Rz_deg(45)   * [arm_half 0 0]')';   % front-right
        motorOffsets(2,:) = (Rz_deg(135)  * [arm_half 0 0]')';   % front-left
        motorOffsets(3,:) = (Rz_deg(-135) * [arm_half 0 0]')';   % rear-left
        motorOffsets(4,:) = (Rz_deg(-45)  * [arm_half 0 0]')';   % rear-right

        % Propellers (flat ellipses)
        Rprop = 0.035;
        t_prop = linspace(0, 2*pi, 40);
        propX = Rprop * cos(t_prop);
```

```
            propY = 0.6 * Rprop * sin(t_prop);
            prop_z_offset = h_motor + 0.008;
            propZ = zeros(size(t_prop)) + prop_z_offset;

            % Plot obstacles
            for j = 1:self.n_objects
                [Xs, Ys, Zs] = sphere(20);
                Xs = self.R_objects(j)*Xs + self.P_objects(1, j);
                Ys = self.R_objects(j)*Ys + self.P_objects(2, j);
                Zs = self.R_objects(j)*Zs + self.P_objects(3, j);

                C = Zs;
                s = surf(Xs, Ys, Zs, C, ...
                         'HandleVisibility','off');
                s.EdgeColor = 'none';
                s.FaceColor = 'interp';
                s.FaceAlpha = 0.4;
                s.LineStyle = 'none';

                text(self.P_objects(1, j), self.P_objects(2, j), ...
                     self.P_objects(3, j) + self.R_objects(j)*0.75, ...
                     sprintf('Obj. %d', j), ...
                     'HorizontalAlignment','center', ...
                     'Interpreter','latex', ...
                     'FontSize',10, 'FontWeight','bold', ...
                     'HandleVisibility','off');
            end

            colormap(parula);
            shading interp;

            % Drone markers & dashed paths
            h_marker = gobjects(self.n_drones,1);
            h_path   = gobjects(self.n_drones,1);

            % Drone body / arms / motors / props
            h_body   = gobjects(self.n_drones,1);
            h_arms   = cell(self.n_drones,4);
            h_motors = cell(self.n_drones,4);
            h_props  = cell(self.n_drones,4);

            for i = 1:self.n_drones
                % Point marker
                h_marker(i) = scatter3(NaN,NaN,NaN, 25, colors(i,:), 'filled', ...
                                       'HandleVisibility','off');

                % Dashed path
                h_path(i)   = plot3(NaN,NaN,NaN, '--', ...
                                    'Color', colors(i,:), ...
                                    'LineWidth', 1.0, ...
```

```
                                    'HandleVisibility','off');

                % Body patch
                h_body(i) = patch('Vertices', bodyVerts, ...
                                  'Faces', bodyFaces, ...
                                  'FaceColor', colors(i,:), ...
                                  'FaceAlpha', 0.30, ...
                                  'EdgeColor', 'none', ...
                                  'HandleVisibility','off');

                % Arm patches
                for a_i = 1:4
                    h_arms{i,a_i} = patch('Vertices', armVerts{a_i}, ...
                                          'Faces', armFaces{a_i}, ...
                                          'FaceColor', colors(i,:), ...
                                          'FaceAlpha', 0.20, ...
                                          'EdgeColor', 'none', ...
                                          'HandleVisibility','off');
                end

                % Motors
                for m_i = 1:4
                    h_motors{i,m_i} = surf(motorX, motorY, motorZ, ...
                                           'FaceColor', colors(i,:), ...
                                           'EdgeColor','none', ...
                                           'FaceAlpha', 0.9, ...
                                           'HandleVisibility','off');
                end

                % Propellers
                for p_i = 1:4
                    h_props{i,p_i} = fill3(propX, propY, propZ, ...
                                           colors(i,:), ...
                                           'FaceAlpha',0.4, ...
                                           'EdgeColor','none', ...
                                           'HandleVisibility','off');
                end
            end

            % Legend handles
            legend_handles = [];
            legend_names   = {};

            % Drone markers (for legend only)
            for i = 1:self.n_drones
                h_leg_marker = scatter3(NaN,NaN,NaN,25,colors(i,:),'filled');
                legend_handles = [legend_handles; h_leg_marker];
                legend_names{end+1} = sprintf('Drone %d', i);
            end

```

```
            % Horizontal legend at bottom
            lgd = legend(legend_handles, legend_names, ...
                         'Interpreter','latex', ...
                         'Orientation','horizontal', ...
                         'NumColumns', self.n_drones, ...
                         'Location','southoutside');
            lgd.FontSize = 10;

            % Timestamp text
            t_handle = annotation(fig_anim, 'textbox', ...
            [0.80 0.92 0.18 0.05], ...   % [x y w h] normalized to figure
            'String','t = 0.00 s', ...
            'Interpreter','latex', ...
            'FontSize',14, ...
            'HorizontalAlignment','right', ...
            'VerticalAlignment','top', ...
            'EdgeColor','none', ...
            'BackgroundColor','none');

            % Start GIF creation
            T = size(x_log{1}, 2);   % number of samples (columns)

            for k = 1:T

                % Update drone trajectory + markers + body model
                for i = 1:self.n_drones
                    xi = x_log{i};

                    % Position
                    px = xi(1,k);
                    py = xi(2,k);
                    pz = xi(3,k);
                    pos = [px py pz];

                    % Orientation (phi, theta, psi)
                    phi   = xi(4,k);   % roll about x
                    theta = xi(5,k);   % pitch about y
                    psi   = xi(6,k);   % yaw about z

                    % Rotation matrices
                    Rx = [1 0 0; ...
                          0 cos(phi) -sin(phi); ...
                          0 sin(phi)  cos(phi)];

                    Ry = [ cos(theta) 0 sin(theta); ...
                           0          1        0; ...
                          -sin(theta) 0 cos(theta)];

                    Rz = [cos(psi) -sin(psi) 0; ...
                          sin(psi)  cos(psi) 0; ...
```

```
                 0         0        1];

        % Total rotation
        R = Rz * Ry * Rx;

        % Update body
        bodyWorld = (R * bodyVerts')' + pos;
        set(h_body(i), 'Vertices', bodyWorld);

        % Update arms
        for a_i = 1:4
            armWorld = (R * armVerts{a_i}')' + pos;
            set(h_arms{i,a_i}, 'Vertices', armWorld);
        end

        % Update motors
        for m_i = 1:4
            offset = motorOffsets(m_i,:);

            Xm = motorX + offset(1);
            Ym = motorY + offset(2);
            Zm = motorZ + offset(3);

            pts_local = [Xm(:)'; Ym(:)'; Zm(:)'];
            pts_world = R * pts_local;

            Xm_w = reshape(pts_world(1,:), size(Xm)) + pos(1);
            Ym_w = reshape(pts_world(2,:), size(Ym)) + pos(2);
            Zm_w = reshape(pts_world(3,:), size(Zm)) + pos(3);

            set(h_motors{i,m_i}, ...
                'XData', Xm_w, ...
                'YData', Ym_w, ...
                'ZData', Zm_w);
        end

        % Update propellers
        for p_i = 1:4
            offset = motorOffsets(p_i,:);

            prop_local = [propX; propY; propZ];
            prop_shift = prop_local + offset';
            prop_world = R * prop_shift + pos';

            set(h_props{i,p_i}, ...
                'XData', prop_world(1,:), ...
                'YData', prop_world(2,:), ...
                'ZData', prop_world(3,:));
        end

```

```
                    % Point marker
                    set(h_marker(i), ...
                         'XData', px, ...
                         'YData', py, ...
                         'ZData', pz);

                    % Dashed path
                    set(h_path(i), ...
                         'XData', xi(1,1:k), ...
                         'YData', xi(2,1:k), ...
                         'ZData', xi(3,1:k));
                end

                % Update time text
                t_handle.String = sprintf('t = %.2f s', (k-1)*self.Ts);

                % Capture frame
                frame = getframe(fig_anim);
                [imind, cm] = rgb2ind(frame2im(frame), 256);

                if k == 1
                    imwrite(imind, cm, gif_name, "gif", ...
                            "Loopcount", inf, "DelayTime", delay);
                else
                    imwrite(imind, cm, gif_name, "gif", ...
                            "WriteMode", "append", "DelayTime", delay);
                end
            end

            close(fig_anim);
        end
    end
end
```