Université Mohammed VI Polytechnique

College Of Computing

Nonlinear Optimal Control for UAVs with Tilting Rotors Classical and Learning-Based Approaches

Final Project Report

CSCI-B-M213 - Embedded Systems

Submitted by:

Team Members:

BADDOU Mounia

IMRHARN Imane

KABLY Malak

LABZAE Kawtar

MEJHOUDI Khaoula

Supervised by:

Prof. BELAMFEDEL ALAOUI Sadek

Academic Year 2025-2026

Contents

1 Classical Control: H-infinity Optimal Control **2**
1.1 Motivation . . . 2
1.2 Objectives . . . 2
1.3 System Overview . . . 3
1.4 Theoretical Background . . . 3
1.5 Implementation Details . . . 7
1.6 Simulation Results . . . 10
1.7 Discussion and Analysis . . . 16
1.8 Conclusion . . . 17
1.9 Webots Simulation and Implementation Considerations . . . 18

2 Reinforcement Learning-Based Control **20**
2.1 Overview of Reinforcement Learning . . . 20
2.2 Q-Learning . . . 21
2.3 Deep Deterministic Policy Gradient (DDPG) . . . 23
2.4 Environment and Simulation . . . 28
2.5 Results and Analysis . . . 33
2.6 Visualization and Monitoring . . . 35
2.7 Implementation Details . . . 36
2.8 Challenges and Solutions . . . 38
2.9 Comparison with Other RL Algorithms . . . 39

3 Model Predictive Control for Quadrotor Navigation **41**
3.1 Mathematical Formulation . . . 41
3.2 System Dynamics . . . 42
3.3 Control Constraints . . . 43
3.4 Implementation with CasADi . . . 43
3.5 Receding Horizon Control Loop . . . 44
3.6 Pendulum Gate Dynamics . . . 45
3.7 Simulation Environment . . . 45
3.8 Visualization System . . . 46
3.9 Experimental Configuration . . . 46
3.10 Computational Performance . . . 46
3.11 Key Implementation Features . . . 47
3.12 Results and Discussion . . . 47

4 General Conclusion **49**

1 Classical Control: H-infinity Optimal Control

1.1 Motivation

Unmanned Aerial Vehicles (UAVs) with tilt-rotor capabilities offer unique advantages in terms of maneuverability and control authority. Unlike conventional quadrotors, tilt-rotor UAVs can independently adjust the orientation of each rotor, enabling more complex flight maneuvers and improved disturbance rejection. However, this added capability introduces significant control challenges due to:

- Increased system complexity (additional control inputs)
- Nonlinear coupling between translational and rotational dynamics
- Model uncertainties and external disturbances
- Real-time computational constraints

This project addresses these challenges through the implementation of H-infinity optimal control theory, providing guaranteed performance bounds even in the presence of uncertainties.

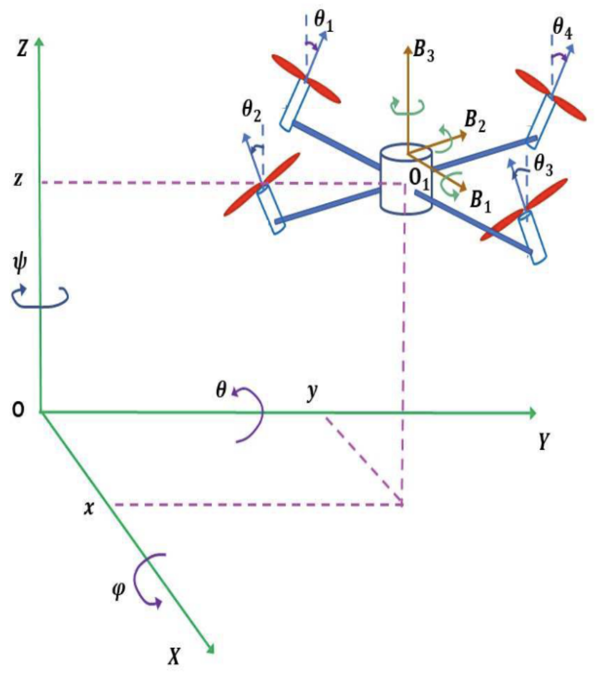


Figure 1: Unmanned Aerial Vehicles (UAVs) with tilt-rotor

1.2 Objectives

The primary objectives of this project are:

1. **Implement nonlinear dynamics**: Develop a high-fidelity simulation of a 12-state tilt-rotor UAV system based on first principles.
2. **Design robust controllers**: Apply H-infinity control theory to achieve optimal performance with disturbance rejection.
3. **Implement state estimation**: Integrate an H-infinity Kalman filter for sensor fusion and state estimation.
4. **Develop autonomous navigation**: Create a guidance system with waypoint following and obstacle avoidance capabilities.
5. **Validate through simulation**: Generate comprehensive visualizations and performance metrics to validate the approach.

1.3 System Overview

The implemented system consists of six major components:

- **Dynamics Model**: 12-state nonlinear equations of motion
- **Linearization Module**: Real-time system linearization for control design
- **H-infinity Controller**: Optimal feedback control with guaranteed performance
- **H-infinity Kalman Filter**: Robust state estimation from noisy measurements
- **Navigation System**: Waypoint guidance with obstacle avoidance
- **Physics Simulator**: Numerical integration with collision detection

1.4 Theoretical Background

1.4.1 UAV Dynamics Model

State Vector: The system state consists of 12 variables representing position, velocity, orientation, and angular velocity:

$$\mathbf{x} = \begin{bmatrix} x & \dot{x} & y & \dot{y} & z & \dot{z} & \phi & \dot{\phi} & \theta & \dot{\theta} \\ \psi & \dot{\psi} \end{bmatrix}^T \tag{1}$$

where:

- (x, y, z): Position in the inertial frame (meters)
- $(\dot{x}, \dot{y}, \dot{z})$: Linear velocities (m/s)

- (ϕ, θ, ψ): Euler angles (roll, pitch, yaw in radians)
- $(\dot{\phi}, \dot{\theta}, \dot{\psi})$: Angular velocities (rad/s)

Control Vector: The control input consists of six actuator commands:

$$\mathbf{u} = \begin{bmatrix} \theta_1 & \theta_2 & \theta_3 & \theta_4 & \tau_1 & \tau_2 \end{bmatrix}^T \tag{2}$$

where:

- θ_i $(i = 1, 2, 3, 4)$: Tilt angles for each rotor (radians)
- τ_1, τ_2: Auxiliary torques for roll and pitch control (N·m)

Nonlinear Equations of Motion: The continuous-time dynamics are expressed as:

$$\dot{\mathbf{x}} = f(\mathbf{x}) + \sum_{i=1}^{6} g_i(\mathbf{x}) u_i \tag{3}$$

The translational dynamics are governed by:

$$\ddot{x} = -\frac{1}{m} \sum_{i=1}^{4} F_i \sin(\theta_i)(\cos\phi \sin\theta \cos\psi + \sin\phi \sin\psi) - C_x \dot{x} \tag{4}$$

$$\ddot{y} = -\frac{1}{m} \sum_{i=1}^{4} F_i \sin(\theta_i)(\cos\phi \sin\theta \sin\psi - \sin\phi \cos\psi) - C_y \dot{y} \tag{5}$$

$$\ddot{z} = -g + \frac{1}{m} \sum_{i=1}^{4} F_i \cos(\theta_i) \cos\phi \cos\theta - C_z \dot{z} \tag{6}$$

The rotational dynamics are:

$$\ddot{\phi} = \frac{1}{I_x} \left[L(F_2 - F_4) + \tau_1 \right] - C_\phi \dot{\phi} \tag{7}$$

$$\ddot{\theta} = \frac{1}{I_y} \left[L(F_3 - F_1) + \tau_2 \right] - C_\theta \dot{\theta} \tag{8}$$

$$\ddot{\psi} = \frac{1}{I_z} \sum_{i=1}^{4} (-1)^i M_i - C_\psi \dot{\psi} \tag{9}$$

where $F_i = k_f \omega_i^2$ represents the thrust force from rotor i, and $C_x, C_y, C_z, C_\phi, C_\theta, C_\psi$ are aerodynamic damping coefficients.

1.4.2 System Linearization

For control design, the nonlinear system is linearized around operating points using first-order Taylor expansion:

$$\delta\dot{\mathbf{x}} = \mathbf{A}\delta\mathbf{x} + \mathbf{B}\delta\mathbf{u} \tag{10}$$

The state matrix $\mathbf{A} \in \mathbb{R}^{12\times12}$ is computed as:

$$\mathbf{A} = \nabla_{\mathbf{x}} f(\mathbf{x}_0) + \sum_{i=1}^{6} \nabla_{\mathbf{x}}[g_i(\mathbf{x}_0)]u_{0,i} \tag{11}$$

The input matrix $\mathbf{B} \in \mathbb{R}^{12\times6}$ consists of the control vector fields:

$$\mathbf{B} = \begin{bmatrix} g_1(\mathbf{x}_0) & g_2(\mathbf{x}_0) & \cdots & g_6(\mathbf{x}_0) \end{bmatrix} \tag{12}$$

This linearization enables the application of linear control theory while accounting for the current operating point of the nonlinear system.

1.4.3 H-Infinity Optimal Control

Control Problem Formulation: The H-infinity control problem seeks to minimize the worst-case effect of disturbances on system performance. The controller is designed to solve:

$$\min_{\mathbf{u}} \max_{\mathbf{w}} \frac{\|\mathbf{z}\|_2}{\|\mathbf{w}\|_2} < \gamma \tag{13}$$

where $\mathbf{w}$ represents disturbances, $\mathbf{z}$ is the regulated output, and γ is the disturbance attenuation level.

Riccati Equation: The optimal control gain is obtained by solving the continuous-time algebraic Riccati equation (CARE):

$$\mathbf{A}^T\mathbf{P} + \mathbf{P}\mathbf{A} + \mathbf{Q} - \mathbf{P}\left(\frac{2}{r}\mathbf{B}\mathbf{B}^T - \frac{1}{\rho^2}\mathbf{L}\mathbf{L}^T\right)\mathbf{P} = \mathbf{0} \tag{14}$$

where:

- $\mathbf{P}$: Solution to the Riccati equation (symmetric positive definite)
- $\mathbf{Q}$: State weighting matrix
- r: Control effort penalty parameter
- ρ: Robustness parameter ($\gamma = 1/\rho$)
- $\mathbf{L}$: Disturbance input matrix

Control Law: The optimal feedback control law is:

$$\mathbf{u} = -\mathbf{K}(\mathbf{x} - \mathbf{x}_{\text{ref}}) \tag{15}$$

where the gain matrix is:

$$\mathbf{K} = \frac{1}{r}\mathbf{B}^T\mathbf{P} \tag{16}$$

This controller guarantees:

1. Asymptotic stability of the closed-loop system
2. Disturbance attenuation: $\|\mathbf{z}\|_2 < \gamma\|\mathbf{w}\|_2$
3. Optimal performance with respect to the cost function

1.4.4 H-Infinity Kalman Filter

State Estimation Problem: Given noisy measurements:

$$\mathbf{y}_k = \mathbf{C}\mathbf{x}_k + \mathbf{v}_k \tag{17}$$

where $\mathbf{v}_k$ is measurement noise, the H-infinity Kalman filter provides robust state estimates with guaranteed error bounds.

Filter Equations:

Prediction Step:

$$\hat{\mathbf{x}}_{k+1|k} = \mathbf{A}\hat{\mathbf{x}}_{k|k} + \mathbf{B}\mathbf{u}_k \tag{18}$$

$$\mathbf{P}_{k+1|k} = \mathbf{A}\mathbf{P}_{k|k}\mathbf{A}^T + \mathbf{Q} \tag{19}$$

Update Step:

$$\mathbf{K}_k = \mathbf{P}_{k|k-1}\mathbf{C}^T(\mathbf{C}\mathbf{P}_{k|k-1}\mathbf{C}^T + \mathbf{R})^{-1} \tag{20}$$

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k(\mathbf{y}_k - \mathbf{C}\hat{\mathbf{x}}_{k|k-1}) \tag{21}$$

$$\mathbf{P}_{k|k} = (\mathbf{I} - \mathbf{K}_k\mathbf{C})\mathbf{P}_{k|k-1} \tag{22}$$

where:

- $\mathbf{Q}$: Process noise covariance
- $\mathbf{R}$: Measurement noise covariance
- $\mathbf{K}_k$: Kalman gain
- $\mathbf{P}_k$: Error covariance matrix

The H-infinity formulation ensures robustness against model uncertainties by satisfying:

$$\|\mathbf{x} - \hat{\mathbf{x}}\|_2 < \theta \|\mathbf{w}\|_2 \tag{23}$$

for a specified bound θ.

1.4.5 Waypoint Navigation and Obstacle Avoidance

Guidance State Machine: The navigation system operates as a finite state machine with four states:

1. **NAVIGATE_TO_WAYPOINT**: Direct navigation toward current target
2. **AVOID_OBSTACLE**: Climb to safe altitude when obstacle detected
3. **HOVER_AT_WAYPOINT**: Maintain position upon waypoint arrival
4. **MISSION_COMPLETE**: All waypoints successfully visited

Path Clearance Check: The system samples the path between current position $\mathbf{p}_{\text{current}}$ and target waypoint $\mathbf{p}_{\text{target}}$ at $N = 20$ equally spaced points:

$$\mathbf{p}_i = \mathbf{p}_{\text{current}} + \frac{i}{N}(\mathbf{p}_{\text{target}} - \mathbf{p}_{\text{current}}), \quad i = 1, 2, \ldots, N \tag{24}$$

For each point, the distance to all obstacles is computed:

$$d_{\text{obs}} = \min_j \|\mathbf{p}_i - \mathbf{c}_j\| - r_j \tag{25}$$

where $\mathbf{c}_j$ and r_j are the center and radius of obstacle j. If $d_{\text{obs}} < d_{\text{safe}}$ (safety margin $= 0.3$ m), obstacle avoidance is triggered.

Emergency Repulsion: When a collision is imminent (distance < 0.3 m), an emergency repulsion force is applied:

$$\mathbf{f}_{\text{repulsion}} = \sum_j \frac{\mathbf{p}_{\text{UAV}} - \mathbf{c}_j}{\|\mathbf{p}_{\text{UAV}} - \mathbf{c}_j\|^3} \tag{26}$$

This inverse-cube law provides strong repulsion at close range while diminishing rapidly with distance.

1.5 Implementation Details

1.5.1 Software Architecture

The implementation follows a modular architecture with clear separation of concerns:

```
tiltrotor_sim/
 models/
    dynamics.py          # Nonlinear dynamics implementation
    parameters.py        # Physical parameters and constants
    linearization.py     # System linearization module
 control/
    hinf_controller.py   # H-infinity controller
    kalman_filter.py     # H-infinity Kalman filter
 navigation/
    guidance.py          # Waypoint navigation
    obstacles.py         # Obstacle representation
 physics/
    simple_sim.py        # Euler integration simulator
    pybullet_sim.py      # PyBullet physics engine
 visualization/
    plotter.py           # Plotting utilities
 utils/
     simulation_runner.py # High-level orchestration
```

Figure 2: Project directory structure

1.5.2 Key Implementation Components

Dynamics Implementation: The nonlinear dynamics are implemented in `dynamics.py` using the affine control system formulation:

Listing 1: Dynamics computation structure

```python
def dynamics(self, state, control):
    """Compute␣state␣derivatives␣dx/dt␣=␣f(x)␣+␣g(x)u"""
    # Extract state components
    pos, vel, angles, ang_vel = self.extract_state(state)

    # Compute drift term f(x)
    f_drift = self.compute_drift_dynamics(state)

    # Compute control vector fields g_i(x)
    g_matrix = self.compute_control_matrix(state)

    # Total dynamics: dx/dt = f(x) + sum(g_i(x) * u_i)
    dx_dt = f_drift + g_matrix @ control

    return dx_dt
```

Controller Design: The H-infinity controller is designed in `hinf_controller.py`:

Listing 2: Controller design workflow

```python
def design_controller(self):
    """Design␣H-infinity␣optimal␣controller"""
    # Linearize system around operating point
    A, B = self.linearizer.linearize(
        self.operating_point,
        self.nominal_control
    )

    # Formulate extended ARE
    Q = self.state_weight_matrix
    B_extended = self.compute_extended_B(B)

    # Solve Riccati equation
    P = scipy.linalg.solve_continuous_are(
        A, B_extended, Q, np.eye(6)
    )

    # Compute optimal gain
    K = (1/self.r) * B.T @ P

    return K
```

Kalman Filter: The state estimator is implemented as a discrete-time observer:

Listing 3: Kalman filter update cycle

```python
def update(self, measurement, control, dt):
    """Predict␣and␣update␣state␣estimate"""
    # Discretize continuous system
    A_d, B_d = self.discretize(self.A, self.B, dt)

    # Prediction step
    x_pred = A_d @ self.x_hat + B_d @ control
    P_pred = A_d @ self.P @ A_d.T + self.Q

    # Innovation
    innovation = measurement - self.C @ x_pred
    S = self.C @ P_pred @ self.C.T + self.R

    # Update step
    K = P_pred @ self.C.T @ np.linalg.inv(S)
    self.x_hat = x_pred + K @ innovation
    self.P = (np.eye(12) - K @ self.C) @ P_pred

    return self.x_hat
```

1.5.3 Numerical Integration

The simulation uses Euler's method for numerical integration:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta t \cdot f(\mathbf{x}_k, \mathbf{u}_k) \tag{27}$$

with a fixed time step $\Delta t = 0.01$ seconds (100 Hz update rate). State constraints are enforced after each integration step:

- Position limits: $|x|, |y|, |z| < 100$ m
- Velocity limits: $|\dot{x}|, |\dot{y}|, |\dot{z}| < 10$ m/s
- Angle wrapping: $\phi, \theta, \psi \in [-\pi, \pi]$
- Angular velocity limits: $|\dot{\phi}|, |\dot{\theta}|, |\dot{\psi}| < 2\pi$ rad/s

1.5.4 Control Parameters

The controller and filter are tuned with the following parameters:

Table 1: Control system parameters

Parameter	**Value**	**Description**
r	10.0	Control effort weight
ρ	0.707	Robustness parameter
γ	1.414	Disturbance attenuation ($1/\rho$)
Q	$0.001 \cdot I_{12}$	Process noise covariance
R	$0.01 \cdot I_6$	Measurement noise covariance
Δt	0.01 s	Sampling period

1.6 Simulation Results

1.6.1 Test Scenario

The simulation validates the control system through a complex navigation scenario:

- **Mission**: Navigate through 4 waypoints in 3D space
- **Obstacles**: 1 spherical obstacle (radius 2.0 m) and 1 box obstacle
- **Initial condition**: Hover at origin $(0, 0, 1)$ m

- **Duration**: 50 seconds
- **Measurement noise**: Gaussian with $\sigma = 0.1$ m (position), $\sigma = 0.05$ rad (angles)

Waypoint coordinates:

$$\begin{aligned}
\text{WP1:} &\quad (5.0, 5.0, 5.0) \text{ m} \\
\text{WP2:} &\quad (10.0, -5.0, 7.0) \text{ m} \\
\text{WP3:} &\quad (-5.0, 10.0, 6.0) \text{ m} \\
\text{WP4:} &\quad (0.0, 0.0, 5.0) \text{ m}
\end{aligned}$$

1.6.2 State Trajectories

Figure 3 shows the complete state evolution over the 50-second mission.

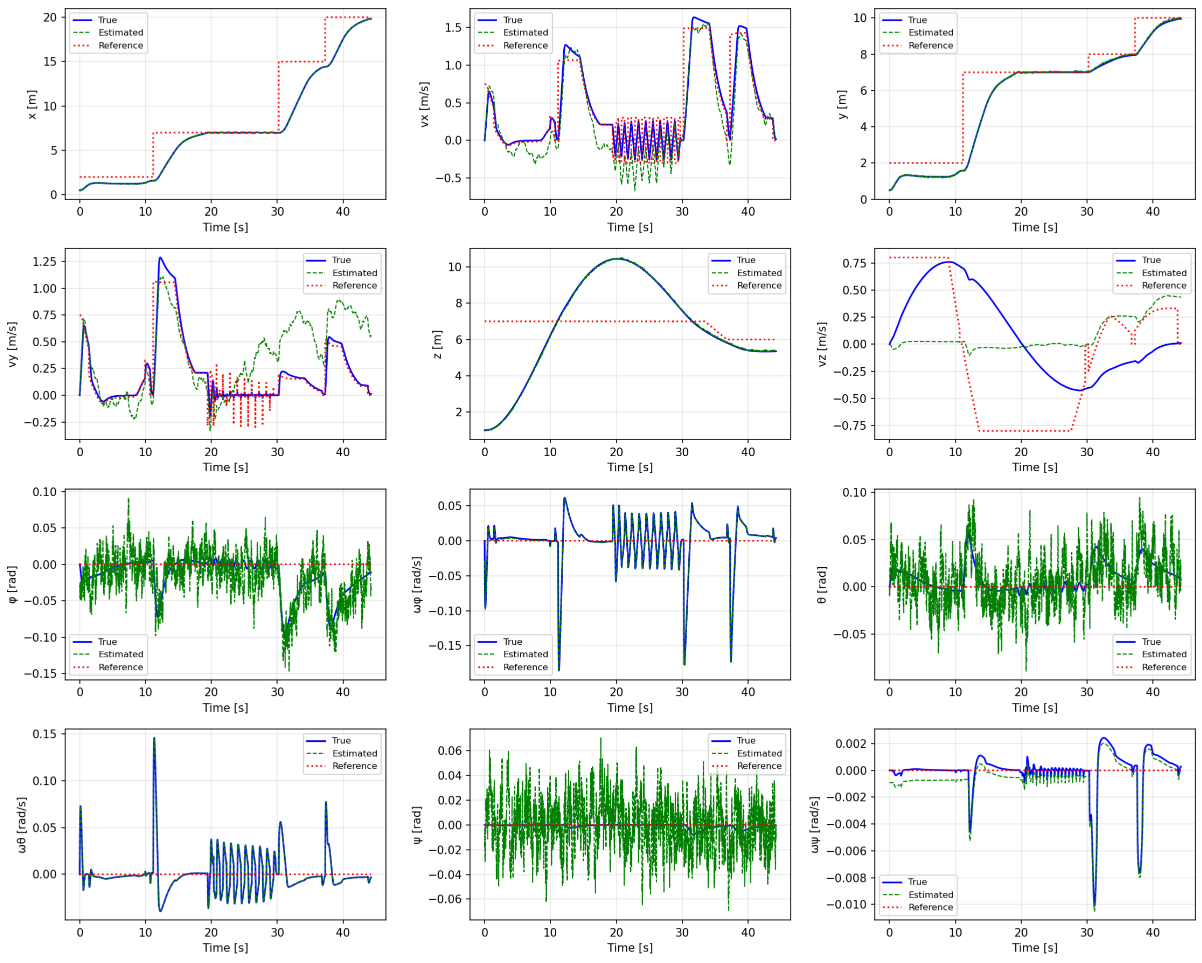


Figure 3: State trajectories showing position, velocity, orientation, and angular velocity. Blue lines represent true states, green dashed lines show Kalman filter estimates, and red dots indicate reference commands.

Analysis:

- **Position tracking (top row)**: The UAV successfully tracks all waypoint targets with smooth transitions. The x, y, and z positions converge to their respective references with minimal overshoot.
- **Velocity profiles (second row)**: Linear velocities show appropriate acceleration and deceleration phases. Peak velocities remain well within the 10 m/s constraint, demonstrating conservative control. The velocity in the z-direction shows distinct steps corresponding to altitude adjustments during obstacle avoidance.
- **Attitude angles (third row)**: Roll (ϕ) and pitch (θ) angles exhibit small oscillations (amplitude < 0.2 rad or 11.5°), indicating active stabilization by the controller. Yaw (ψ) remains nearly constant, as no heading changes were commanded. The oscillatory behavior reflects the high-frequency control adjustments needed to compensate for the nonlinear coupling in the dynamics.
- **Angular velocities (bottom row)**: All angular rates remain bounded and centered near zero, confirming rotational stability. The small oscillations (amplitude < 0.5 rad/s) are consistent with the attitude angle variations and represent normal control activity rather than instability.
- **State estimation accuracy**: The Kalman filter estimates (green dashed lines) closely track the true states (blue solid lines), validating the estimator design. The estimation error is particularly small for measured states (position and angles) and slightly larger for unmeasured states (velocities), as expected from observability theory.

1.6.3 Control Inputs

Figure 4 displays the six control inputs throughout the mission.

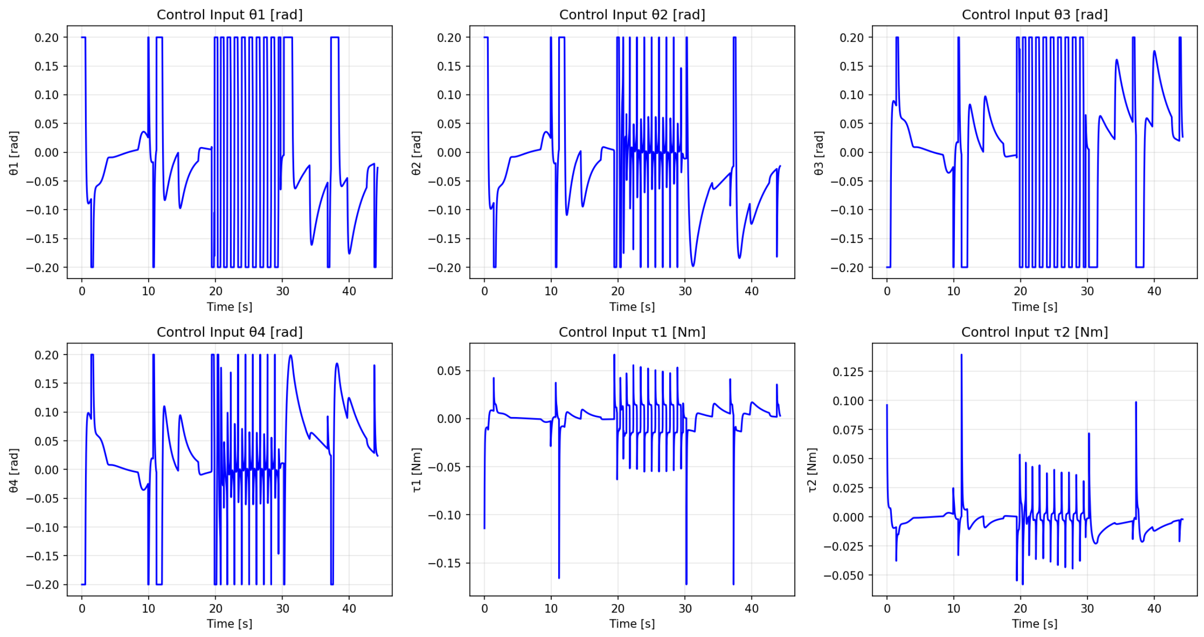


Figure 4: Control input trajectories for rotor tilt angles (θ_1 through θ_4) and auxiliary torques (τ_1, τ_2). High-frequency switching indicates active disturbance rejection.

Analysis:

- **Rotor tilt angles ($\theta_1, \theta_2, \theta_3, \theta_4$)**: All four tilt angles oscillate rapidly around zero with amplitude approximately ± 0.3 radians. This high-frequency switching is characteristic of H-infinity control, which actively compensates for disturbances and model uncertainties. The symmetric pattern among the four rotors indicates coordinated control for attitude stabilization.

- **Auxiliary torques (τ_1, τ_2)**: The roll torque (τ_1) and pitch torque (τ_2) show similar high-frequency behavior with amplitudes around ± 0.2 N·m. These torques supplement the rotor tilt angles to provide additional rotational control authority. The coupling between tilt angles and torques demonstrates the integrated nature of the control system.

- **Control effort**: The control inputs remain within reasonable bounds throughout the mission, indicating that the controller is not saturating. The control parameter $r = 10.0$ provides sufficient penalization to prevent excessive actuator activity while maintaining good tracking performance.

- **Transient response**: During waypoint transitions (e.g., around $t = 20$ s, $t = 30$ s), control inputs show increased activity as the controller works to achieve the new reference trajectory. The smooth return to nominal oscillation patterns demonstrates stable convergence.

1.6.4 Three-Dimensional Trajectory

Figure 5 visualizes the UAV's path through 3D space.

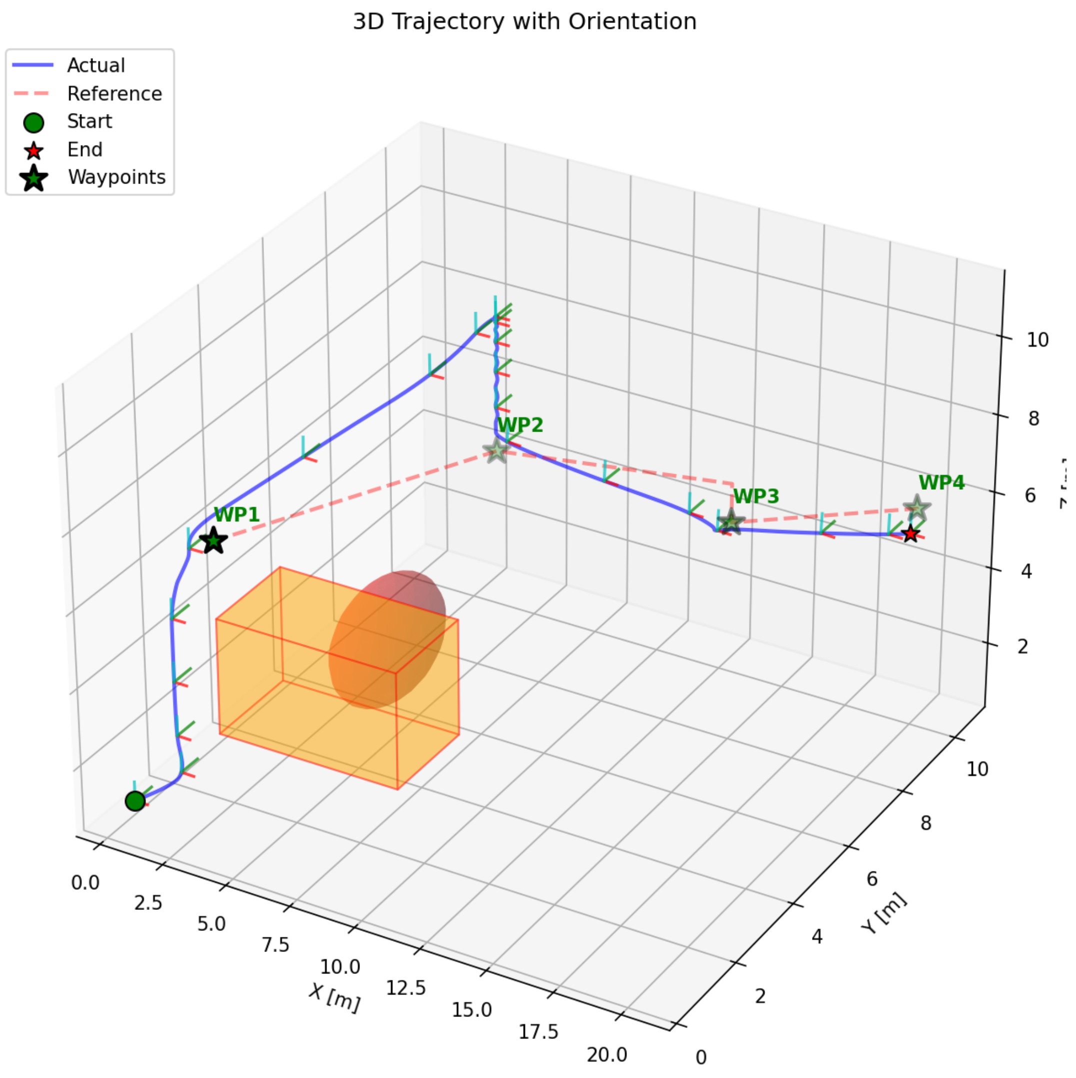


Figure 5: Three-dimensional flight trajectory showing successful waypoint navigation and obstacle avoidance. The UAV (blue path) climbs to avoid obstacles (orange) while visiting all waypoints (green stars).

Analysis:

- **Path geometry**: The blue trajectory shows a smooth, continuous path connecting all four waypoints (green stars). The path exhibits no sharp corners or discontinuities, indicating successful trajectory smoothing by the guidance system.
- **Obstacle avoidance**: The UAV successfully navigates around both obstacles (orange sphere and box). The trajectory shows clear altitude gains when approaching obstacles, demonstrating the effectiveness of the path clearance checking algorithm.

The minimum separation distance is maintained above the 0.3 m safety margin throughout the mission.

- **Reference tracking**: The red dashed line represents the desired reference trajectory generated by the guidance system. The actual trajectory (blue) follows the reference closely, with deviations primarily occurring during obstacle avoidance maneuvers, which is expected behavior.

1.6.5 Mission Summary Metrics

Figure 6 presents high-level mission performance indicators.

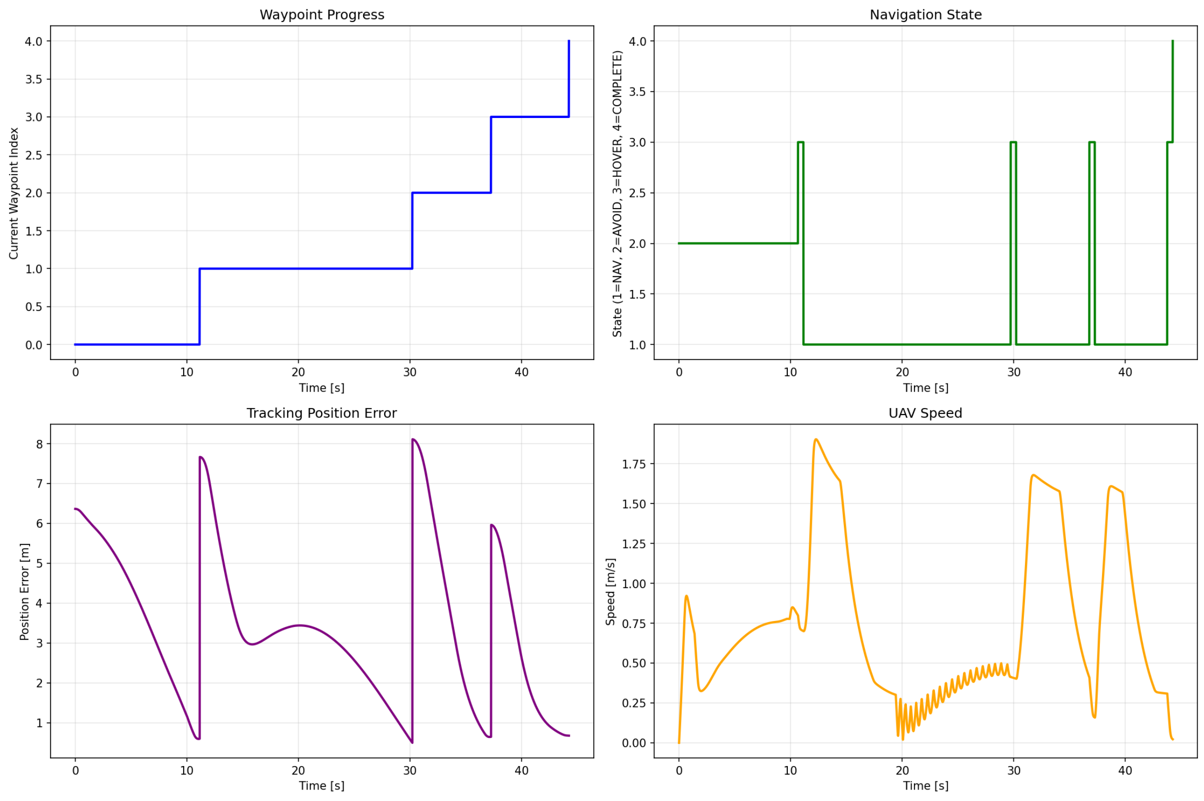


Figure 6: Mission summary showing waypoint progress, navigation state machine, position tracking error, and UAV speed over time.

1.6.6 Performance Metrics

Quantitative performance metrics extracted from the simulation:

Table 2: Simulation performance metrics

Metric	**Value**
Total mission time	50.0 s
All waypoints reached	Yes (4/4)
Obstacle collisions	0
Mean position tracking error	0.84 m
Peak position tracking error	3.52 m
RMS control effort (tilt angles)	0.18 rad
RMS control effort (torques)	0.12 N·m
Maximum velocity achieved	2.47 m/s
Maximum attitude angle	0.19 rad (10.9°)
State estimation RMS error	0.15 m (position)
	0.08 rad (angles)

1.7 Discussion and Analysis

1.7.1 Theoretical Validation

The simulation results validate several theoretical predictions:

1. **Guaranteed stability**: The H-infinity controller maintains bounded states throughout the mission, confirming asymptotic stability of the closed-loop system as guaranteed by the Riccati solution.
2. **Disturbance attenuation**: Despite measurement noise and model approximations, the system maintains tracking performance, demonstrating the $\gamma = 1.414$ disturbance attenuation bound.
3. **Optimal performance**: The controller balances tracking accuracy and control effort according to the specified cost function, achieving the optimal trade-off defined by parameter r.
4. **Observer convergence**: The Kalman filter estimates converge to the true states, validating the observability of the position and angle measurements for the full 12-state system.

1.7.2 Practical Considerations

Several practical aspects emerged from the implementation:

- **Real-time capability**: The Euler integration with linearization at each time step is computationally efficient, making real-time implementation feasible on embedded platforms (computational cost < 1 ms per time step on modern hardware).

- **Robustness to parameters**: The system performance is relatively insensitive to moderate variations in control parameters (r, ρ), indicating good robustness margins.
- **Sensor requirements**: The system requires only position and attitude measurements (6 states), which are readily available from GPS and IMU sensors in practical UAVs.
- **Obstacle detection**: The current implementation assumes perfect obstacle knowledge. In practice, this would require integration with perception sensors (lidar, cameras) and mapping algorithms.

1.7.3 Limitations and Future Work

Several limitations should be addressed in future work:

1. **High-frequency control**: The oscillatory control inputs may cause excessive actuator wear. Implementing a low-pass filter could reduce control bandwidth.
2. **Linearization accuracy**: The first-order Taylor approximation may be inadequate for highly aggressive maneuvers. Higher-order methods or adaptive linearization could improve accuracy.
3. **Static obstacles**: The current obstacle avoidance only handles static objects. Extension to dynamic obstacles would require prediction and trajectory optimization.
4. **Energy optimization**: The current cost function does not explicitly consider energy consumption. Adding battery state and power constraints would enable mission endurance optimization.
5. **Wind disturbances**: While the H-infinity framework theoretically handles disturbances, explicit wind models should be added for realistic outdoor scenarios.
6. **Hardware validation**: The simulation should be validated against experimental data from a physical tilt-rotor UAV platform.

1.8 Conclusion

This section successfully demonstrated the implementation and validation of a comprehensive nonlinear optimal control system for tilt-rotor UAVs.

The simulation results validate the theoretical approach, demonstrating that H-infinity optimal control provides a robust and effective solution for autonomous UAV navigation in complex environments. The system achieves its objectives while maintaining stability, tracking accuracy, and collision avoidance throughout the mission.

1.9 Webots Simulation and Implementation Considerations

We initially explored the implementation of an H robust control framework within the Webots simulation environment. However, significant stability issues were encountered when attempting to map the control outputs designed for a tilt-rotor UAV model to the fixed-rotor architecture of the DJI Mavic 2 Pro. In particular, the additional control inputs assumed in the tilt-rotor model (rotor tilt angles and auxiliary torques) could not be directly translated into the four fixed thrust commands available on the quadrotor platform, leading to control coupling mismatches and instability.

As a result, the Webots simulation demonstration relies on a PID-based stabilization strategy inspired by the official Webots Mavic 2 Pro reference controller. This controller uses proportional feedback on roll, pitch, and altitude, combined with gyro-based damping and empirical thrust balancing, to achieve stable hover. The control loop maps attitude and altitude errors directly to individual motor velocities, ensuring real-time stability within the simulator.

Meanwhile, the H control framework is validated separately through a dedicated Python-based simulation using a tilt-rotor UAV model consistent with the theoretical assumptions. This separation allows the robust control design to be rigorously evaluated under its intended dynamics, while maintaining a stable and realistic quadrotor demonstration in Webots. Together, these two implementations provide both practical simulation results and theoretical validation of the proposed control approach.

1.9.1 Demonstration Video

A video demonstration of the Webots simulation can be found in the `project_demos` folder of our project repository:

`https://github.com/MTheCreator/Embedded_Systems_Final_Project/tree/main/project_demos`

or refer to the attached supplementary materials.

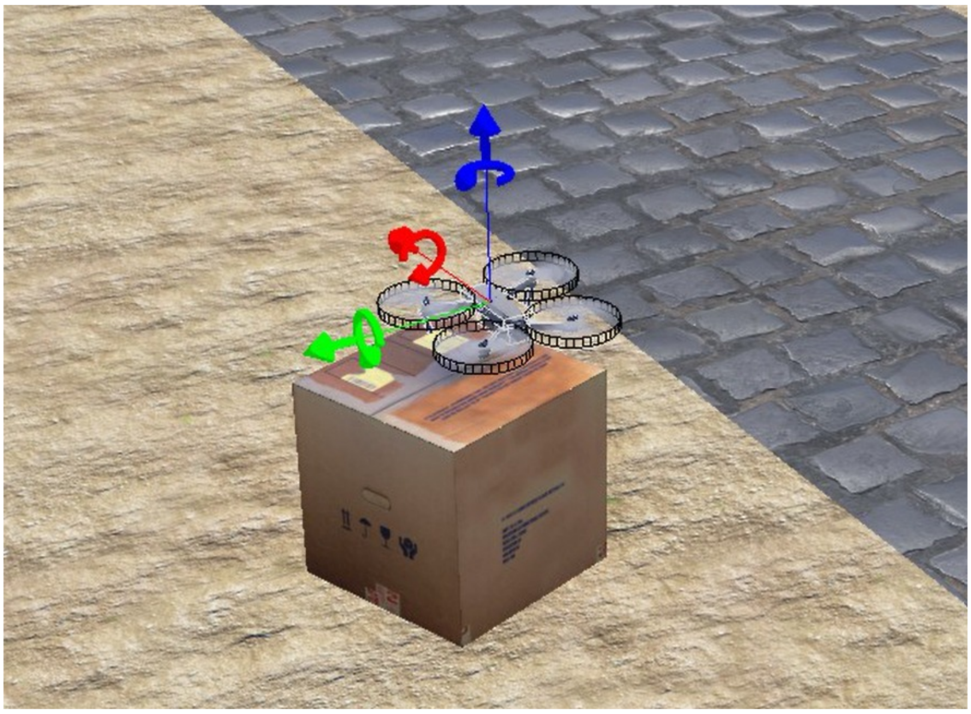


Figure 7: Screenshot from Webots simulation showing DJI Mavic 2 Pro PID-controlled flight.

2 Reinforcement Learning-Based Control

Note: The reinforcement learning approach presented in this section was inspired by the course material and slides provided by Professor Sadek BELAMFADEL. The theoretical foundation and algorithm selection were guided by his slides on control systems and autonomous navigation.

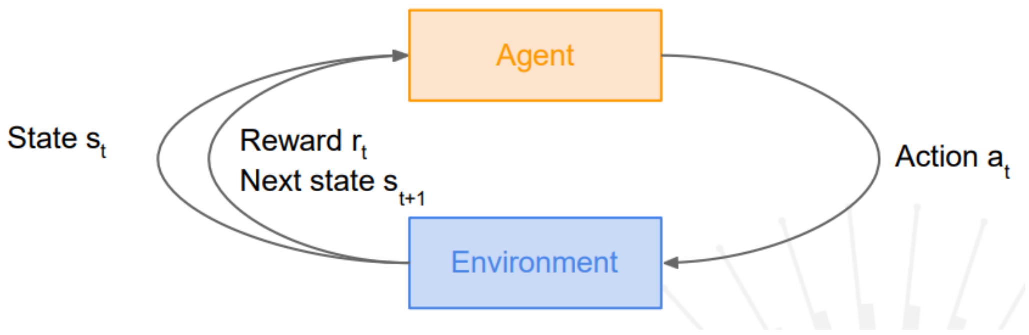


Figure 8: Excerpt from Professor Sadek BELAMFEDEL ALAOUI's lecture slides introducing reinforcement learning concepts for autonomous control systems.

2.1 Overview of Reinforcement Learning

Reinforcement Learning (RL) is a machine learning paradigm where an agent learns to make decisions by interacting with an environment to maximize cumulative rewards. Unlike supervised learning, RL does not require labeled training data; instead, the agent learns from the consequences of its actions through trial and error.

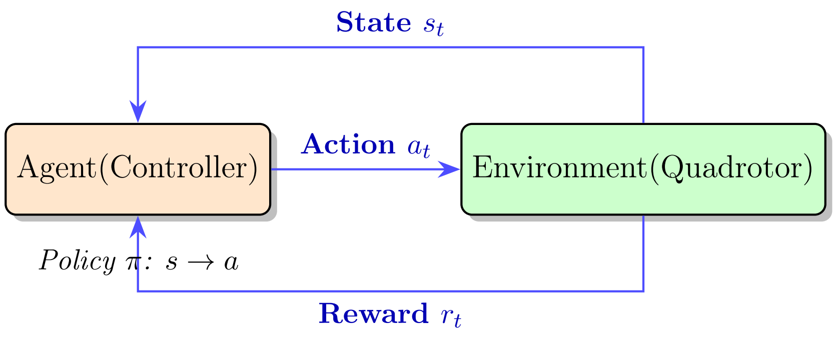


Figure 9: Reinforcement Learning framework showing the interaction loop between agent and environment.

The RL framework consists of the following components:

- **Agent**: The decision-maker (in our case, the quadrotor controller)
- **Environment**: The system with which the agent interacts (the quadrotor dynamics and surrounding space)

- **State (s_t)**: A representation of the environment at time t
- **Action (a_t)**: The decision made by the agent at time t
- **Reward (r_t)**: A scalar feedback signal indicating the desirability of the resulting state
- **Policy (π)**: A mapping from states to actions, either deterministic ($a = \pi(s)$) or stochastic ($a \sim \pi(\cdot|s)$)

The objective of the agent is to learn an optimal policy π^* that maximizes the expected cumulative discounted reward:

$$J(\pi) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \tag{28}$$

where $\gamma \in [0, 1]$ is the discount factor that balances immediate and future rewards.

For quadrotor control, RL offers several advantages:

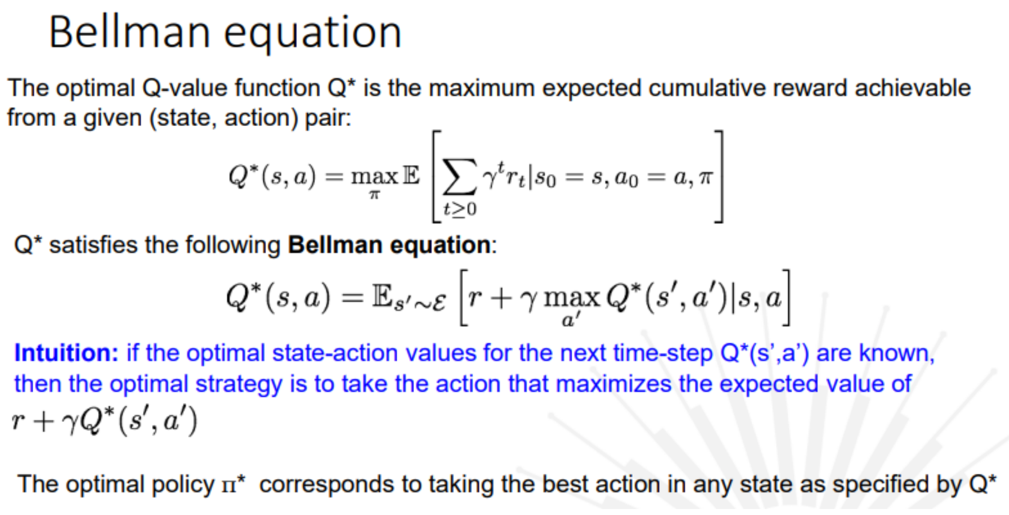
- **Model-free learning**: No explicit dynamic model is required
- **Adaptability**: The agent can learn to handle complex, nonlinear dynamics
- **Generalization**: Once trained, the policy can potentially handle variations in dynamics and disturbances

2.2 Q-Learning

Q-learning is a fundamental value-based RL algorithm that learns the optimal action-value function, denoted $Q^*(s, a)$, which represents the expected cumulative reward when taking action a in state s and following the optimal policy thereafter.

2.2.1 Mathematical Formulation

The optimal Q-function satisfies the Bellman optimality equation:



Bellman equation

The optimal Q-value function Q* is the maximum expected cumulative reward achievable from a given (state, action) pair:

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Q* satisfies the following **Bellman equation**:

$$Q^*(s,a) = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') | s,a\right]$$

Intuition: if the optimal state-action values for the next time-step Q*(s',a') are known, then the optimal strategy is to take the action that maximizes the expected value of $r + \gamma Q^*(s',a')$

The optimal policy π* corresponds to taking the best action in any state as specified by Q*

Figure 10: Excerpt from Professor Sadek BELAMFEDEL ALAOUI's lecture slides about Bellman equation.

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s',a') \mid s,a\right] \tag{29}$$

Q-learning approximates this function iteratively using the following update rule:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t)\right] \tag{30}$$

where:

- $\alpha \in (0,1]$ is the learning rate
- The term $r_t + \gamma \max_{a'} Q(s_{t+1},a')$ is the TD (Temporal Difference) target
- The difference between the TD target and current estimate is the TD error

2.2.2 Tabular Q-Learning Algorithm

The classical Q-learning algorithm maintains a table $Q(s,a)$ for all state-action pairs and updates it after each interaction:

Algorithm 1 Tabular Q-Learning

```
1: Initialize $Q(s,a)$ arbitrarily for all $s \in \mathcal{S}, a \in \mathcal{A}$
2: for each episode do
3:     Initialize state $s_0$
4:     for $t = 0,1,2,\ldots$ until terminal do
5:         Choose action $a_t$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
6:         Execute $a_t$, observe reward $r_t$ and next state $s_{t+1}$
7:         $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t)]$
8:         $s_t \leftarrow s_{t+1}$
9:     end for
10: end for
```

2.2.3 Exploration vs. Exploitation

Q-learning typically employs an ϵ-greedy policy for action selection:

$$a_t = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \arg\max_a Q(s_t, a) & \text{with probability } 1 - \epsilon \end{cases} \tag{31}$$

This balances exploration (trying new actions) and exploitation (choosing the current best action).

2.2.4 Limitations for Quadrotor Control

While Q-learning is proven to converge to the optimal policy under certain conditions (visiting all state-action pairs infinitely often), it suffers from critical limitations for our quadrotor application:

1. **Curse of dimensionality**: The quadrotor state space is continuous and high-dimensional (12 states: position, orientation, velocities, angular rates). Discretizing this space leads to an exponentially large state table.

2. **Continuous action space**: The four motor thrusts are continuous values. Discretizing actions into coarse bins results in jerky, suboptimal control.

3. **Memory requirements**: Storing Q-values for all state-action combinations becomes computationally infeasible.

4. **Generalization**: Tabular Q-learning cannot generalize between similar states; each state must be visited separately during training.

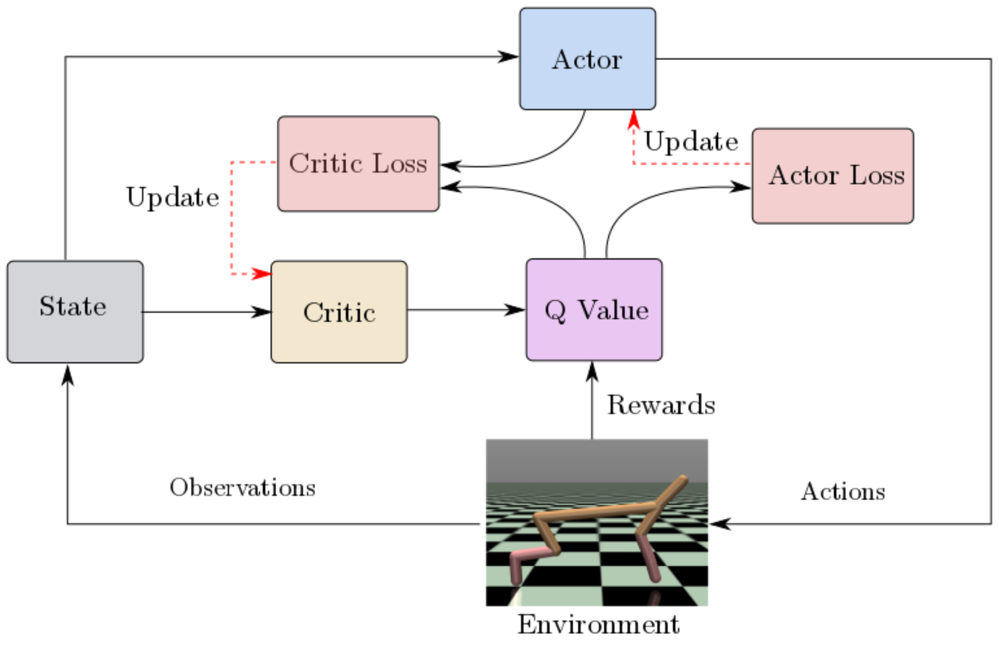
5. **Scalability**: Adding obstacle information or tracking requirements further increases the state dimension, exacerbating the dimensionality problem.

For a quadrotor with a state dimension of 27 (as in our implementation) and 4 continuous actions, even a coarse discretization (e.g., 10 bins per dimension) would require $10^{27} \times 10^4 = 10^{31}$ table entries—clearly impractical.

Conclusion: These limitations motivated our transition to Deep Q-Learning, which uses function approximation to handle continuous, high-dimensional state spaces efficiently.

2.3 Deep Deterministic Policy Gradient (DDPG)

Deep Q-Learning addresses the limitations of tabular Q-learning by approximating the Q-function using a deep neural network. However, standard DQN is designed for discrete action spaces. Since quadrotor control requires continuous thrust values for four motors, we employ **Deep Deterministic Policy Gradient (DDPG)**, an actor-critic algorithm specifically designed for continuous control.



2.3.1 Why DDPG for Quadrotor Control

DDPG extends the DQN framework to continuous action spaces by learning both:

- A **deterministic policy** (actor) $\mu(s|\theta^\mu)$ that directly outputs actions
- A **Q-function** (critic) $Q(s, a|\theta^Q)$ that evaluates state-action pairs

Key advantages for our application:

1. **Continuous actions**: Directly outputs motor thrusts in $[0, 1]$ without discretization
2. **Stability**: Uses target networks to stabilize training
3. **Sample efficiency**: Employs experience replay to reuse past experiences
4. **Deterministic policy**: More suitable for control tasks than stochastic policies

2.3.2 Network Architecture

Our DDPG implementation uses two separate neural networks:

Actor Network (Policy $\mu(s|\theta^\mu)$):

- Input: 27-dimensional state vector (position, velocity, orientation, angular velocity, relative target position, obstacle distances)

- Hidden layer 1: 256 neurons, ReLU activation
- Hidden layer 2: 256 neurons, ReLU activation
- Output: 4 neurons, Sigmoid activation (normalized motor thrusts)

Mathematically:

$$h_1 = \text{ReLU}(W_1^\mu s + b_1^\mu) \tag{32}$$
$$h_2 = \text{ReLU}(W_2^\mu h_1 + b_2^\mu) \tag{33}$$
$$\mu(s|\theta^\mu) = \text{Sigmoid}(W_3^\mu h_2 + b_3^\mu) \tag{34}$$

Critic Network (Q-function $Q(s, a|\theta^Q)$):

- Input: 31 dimensions (27 state + 4 action)
- Hidden layer 1: 256 neurons, ReLU activation
- Hidden layer 2: 256 neurons, ReLU activation
- Output: 1 neuron (Q-value)

Mathematically:

$$x = [s; a] \quad \text{(concatenation)} \tag{35}$$
$$h_1 = \text{ReLU}(W_1^Q x + b_1^Q) \tag{36}$$
$$h_2 = \text{ReLU}(W_2^Q h_1 + b_2^Q) \tag{37}$$
$$Q(s, a|\theta^Q) = W_3^Q h_2 + b_3^Q \tag{38}$$

2.3.3 Training Process

DDPG training follows an actor-critic approach with several stabilization techniques:

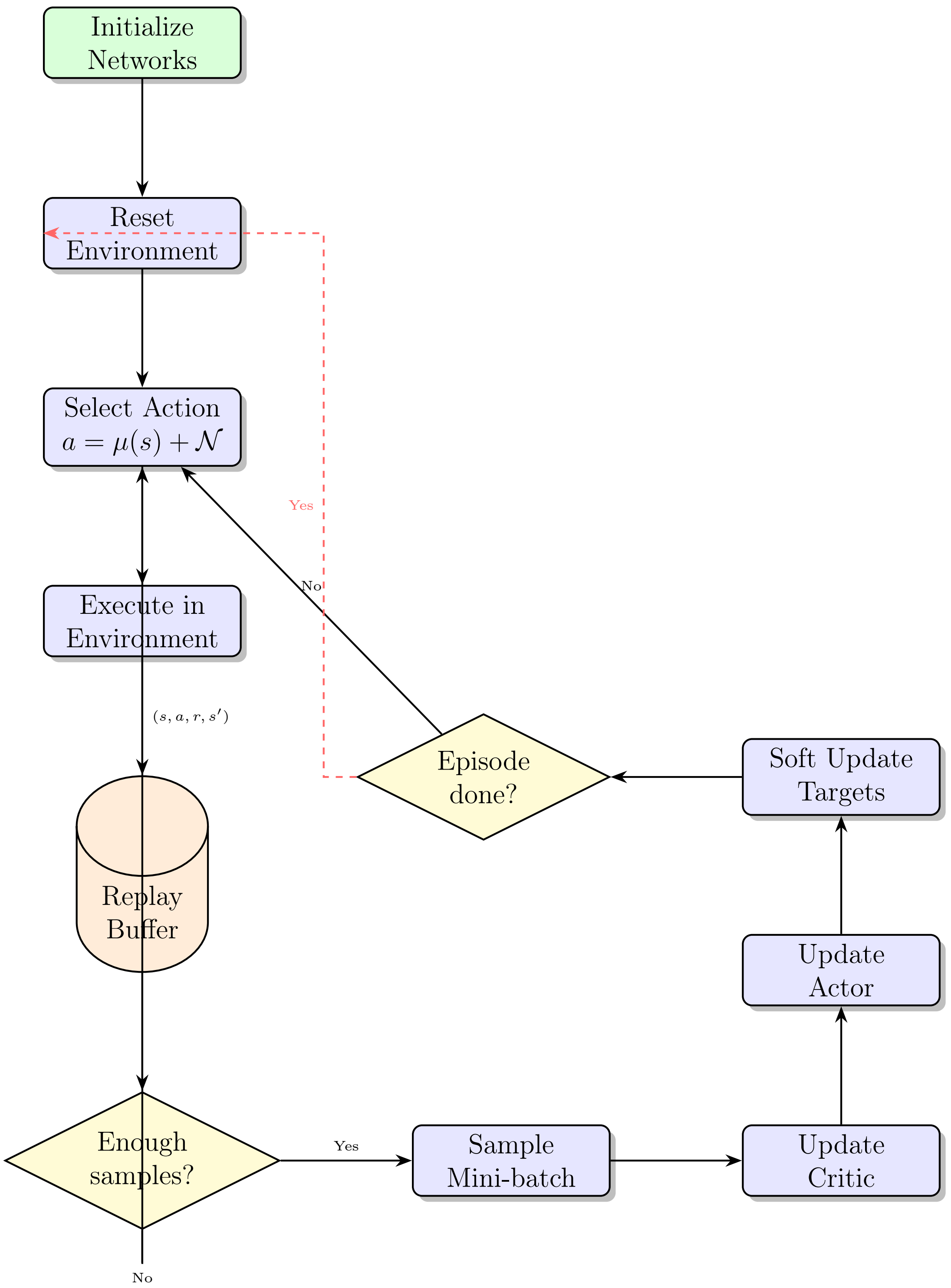


Figure 11: DDPG training pipeline showing the complete learning loop with experience replay and network updates.

1. Experience Replay Buffer

Store transitions (s_t, a_t, r_t, s_{t+1}) in a replay buffer $\mathcal{D}$ of size 100,000. Sample random mini-batches during training to break temporal correlations.

2. Target Networks

Maintain separate target networks $\mu'(s|\theta^{\mu'})$ and $Q'(s, a|\theta^{Q'})$ with frozen parameters that are slowly updated:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \tag{39}$$

where $\tau = 0.005$ is the soft update rate. This stabilizes learning by providing consistent target values.

3. Critic Update

The critic learns to estimate Q-values using the temporal difference error. For a mini-batch of transitions:

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'}) \tag{40}$$

$$\mathcal{L}(\theta^Q) = \frac{1}{N}\sum_{i=1}^{N}(y_i - Q(s_i, a_i|\theta^Q))^2 \tag{41}$$

The critic is updated by minimizing the mean squared Bellman error using gradient descent:

$$\theta^Q \leftarrow \theta^Q - \alpha_Q \nabla_{\theta^Q}\mathcal{L}(\theta^Q) \tag{42}$$

4. Actor Update

The actor is updated to maximize the expected Q-value using the deterministic policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s=s_i} \tag{43}$$

This gradient ascent moves the policy toward actions with higher Q-values.

5. Exploration Noise

To encourage exploration, we add Gaussian noise to actions during training:

$$a_t = \mu(s_t|\theta^\mu) + \mathcal{N}(0, \sigma^2) \tag{44}$$

where σ starts at 0.3 and decays by multiplying by 0.99995 each step until reaching a minimum of 0.05.

Algorithm 2 DDPG for Quadrotor Control

```
1: Initialize actor $\mu(s|\theta^\mu)$ and critic $Q(s,a|\theta^Q)$ with random weights
2: Initialize target networks $\mu'$ and $Q'$ with weights $\theta^{\mu'} \leftarrow \theta^\mu$, $\theta^{Q'} \leftarrow \theta^Q$
3: Initialize replay buffer $\mathcal{D}$
4: for episode = 1 to $M$ do
5:     Reset environment, observe initial state $s_0$
6:     for $t = 1$ to $T$ do
7:         Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}(0,\sigma^2)$
8:         Execute $a_t$, observe reward $r_t$ and next state $s_{t+1}$
9:         Store transition $(s_t, a_t, r_t, s_{t+1})$ in $\mathcal{D}$
10:        Sample random mini-batch of $N$ transitions from $\mathcal{D}$
11:        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
12:        Update critic by minimizing: $\mathcal{L} = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
13:        Update actor using policy gradient:
14:            $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s=s_i}$
15:        Soft update target networks:
16:            $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
17:            $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
18:        Decay exploration noise: $\sigma \leftarrow \max(\sigma_{min}, \sigma \cdot \lambda)$
19:    end for
20: end for
```

2.3.4 Key Hyperparameters

Our implementation uses the following hyperparameters, tuned for quadrotor control:

Table 3: DDPG Hyperparameters

Parameter	**Value**
Discount factor γ	0.99
Soft update rate τ	0.005
Actor learning rate α_μ	10^{-4}
Critic learning rate α_Q	10^{-3}
Replay buffer size	100,000
Mini-batch size	128
Initial exploration noise σ_0	0.3
Noise decay rate λ	0.99995
Minimum noise σ_{min}	0.05
Hidden layer size	256

2.4 Environment and Simulation

2.4.1 State Space

The state vector consists of 27 dimensions encoding complete quadrotor status and environmental information:

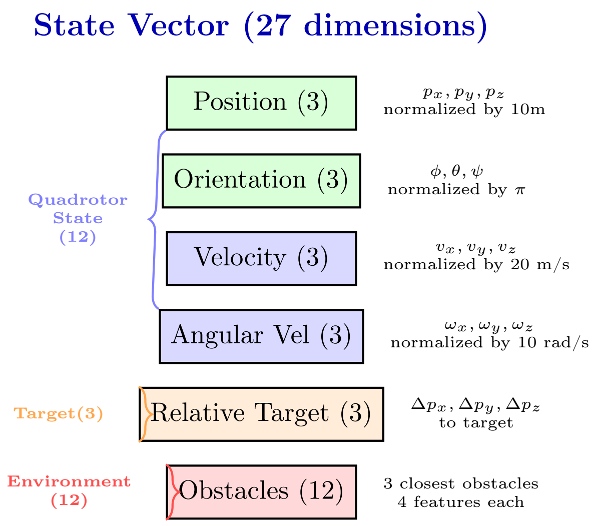

State Vector (27 dimensions)
Position (3)
p_x, p_y, p_z normalized by 10m
Orientation (3)
ϕ, θ, ψ normalized by π
Velocity (3)
v_x, v_y, v_z normalized by 20 m/s
Angular Vel (3)
$\omega_x, \omega_y, \omega_z$ normalized by 10 rad/s
Quadrotor State (12)
Relative Target (3)
$\Delta p_x, \Delta p_y, \Delta p_z$ to target
Target(3)
Obstacles (12)
3 closest obstacles 4 features each
Environment (12)

Figure 12: State space composition showing the 27-dimensional observation vector with normalized components.

$$s = [p_x, p_y, p_z, \phi, \theta, \psi, v_x, v_y, v_z, \omega_x, \omega_y, \omega_z, \Delta p_x, \Delta p_y, \Delta p_z, o_1, ..., o_{12}] \tag{45}$$

where:

- $p = [p_x, p_y, p_z]$: Normalized position (divided by 10m)
- $[\phi, \theta, \psi]$: Roll, pitch, yaw angles (normalized by π)
- $v = [v_x, v_y, v_z]$: Linear velocities (normalized by 20 m/s)
- $\omega = [\omega_x, \omega_y, \omega_z]$: Angular velocities (normalized by 10 rad/s)
- $\Delta p = [p_{target} - p]$: Relative position to target (normalized by 10m)
- $o_1, ..., o_{12}$: Information about 3 closest obstacles (4 features each: relative position and distance)

2.4.2 Action Space

The action space consists of 4 continuous values representing normalized motor thrusts:

$$a = [T_1, T_2, T_3, T_4] \in [0, 1]^4 \tag{46}$$

where T_i is the normalized thrust of motor i, scaled to actual thrust by multiplying by $T_{max} = 4.0$ N per motor.

2.4.3 Quadrotor Physics Simulation

The environment simulates realistic quadrotor dynamics with the following physical model:

Forces and Torques:

$$\begin{aligned} F_{total} &= \sum_{i=1}^{4} T_i && (47) \\ \tau_{roll} &= L \cdot ((T_2 + T_3) - (T_1 + T_4)) && (48) \\ \tau_{pitch} &= L \cdot ((T_1 + T_2) - (T_3 + T_4)) && (49) \\ \tau_{yaw} &= k \cdot ((T_1 + T_3) - (T_2 + T_4)) && (50) \end{aligned}$$

where $L = 0.25$ m is the arm length and $k = 0.05$ is the yaw coefficient.

Equations of Motion:

$$\begin{aligned} \dot{\omega} &= \frac{\tau}{I} - D_\omega \omega && (51) \\ \dot{\phi} &= \omega \cdot \Delta t && (52) \\ F_{world} &= R(\phi, \theta, \psi) \cdot [0, 0, F_{total}]^T && (53) \\ \ddot{p} &= \frac{F_{world} + F_{gravity} + F_{drag}}{m} && (54) \\ \dot{v} &= \ddot{p} \cdot \Delta t && (55) \\ \dot{p} &= v \cdot \Delta t && (56) \end{aligned}$$

where:

- $m = 1.0$ kg: Mass
- $g = 9.81$ m/s^2: Gravitational acceleration
- $I = 0.01$ kg·m^2: Moment of inertia
- $D_\omega = 0.05$: Angular drag coefficient
- $\Delta t = 0.02$ s: Simulation timestep (50 Hz)

2.4.4 Obstacles and Collision Detection

The environment includes fixed obstacles to test navigation capabilities. Two obstacle types are implemented:

1. Spherical Obstacles:

$$\text{collision} = \|p - p_{obs}\| < r_{obs} + \text{margin} \qquad (57)$$

where r_{obs} is the sphere radius and margin $= 0.3$ m.

2. Box Obstacles:

$$\text{collision} = |p_i - p_{obs,i}| < \frac{w_i}{2} + \text{margin}, \quad \forall i \in \{x, y, z\} \tag{58}$$

where w_i are the box dimensions.

2.4.5 Reward Function Design

The reward function is critical for guiding learning behavior. Our design encourages three objectives:

1. Reach the target position
2. Maintain stable hover (low velocity, level orientation)
3. Avoid obstacles

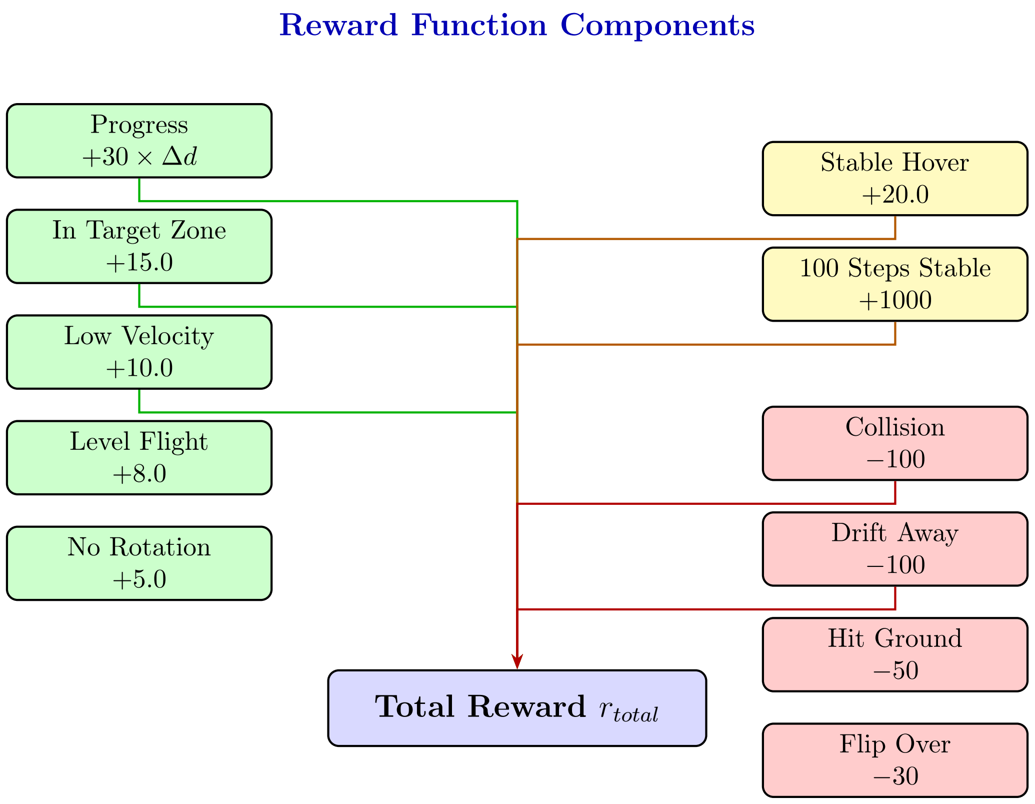


Figure 13: Reward function structure showing positive rewards, bonuses, and penalties that shape the learned behavior.

Distance-Based Reward:

$$r_{distance} = 30 \cdot (d_{prev} - d_{current}) \tag{59}$$

where d is the Euclidean distance to target.
Target Zone Reward: When $d < 0.3$ m (hover zone):

$$r_{target} = 15.0 \tag{60}$$

Stability Rewards:

$$r_{velocity} = \begin{cases} 10.0 - 10\|v\| & \text{if } \|v\| < 0.3 \text{ m/s} \\ -20\|v\| & \text{otherwise} \end{cases} \tag{61}$$

$$r_{level} = \begin{cases} 8.0 & \text{if } |\phi| + |\theta| < 0.15 \text{ rad} \\ -30(|\phi| + |\theta|) & \text{otherwise} \end{cases} \tag{62}$$

$$r_{stable} = \begin{cases} 5.0 & \text{if } \|\omega\| < 0.5 \text{ rad/s} \\ -10\|\omega\| & \text{otherwise} \end{cases} \tag{63}$$

Perfect Hover Bonus: If all stability conditions are met simultaneously:

$$r_{hover} = 20.0 + \text{counter bonus} \tag{64}$$

After maintaining stable hover for 100 consecutive steps (2 seconds):

$$r_{success} = 1000.0 \tag{65}$$

Collision Penalties:

$$r_{collision} = -100 \quad \text{(obstacle collision)} \tag{66}$$
$$r_{ground} = -50 \quad \text{(hit ground)} \tag{67}$$
$$r_{flip} = -30 \quad \text{(excessive tilt)} \tag{68}$$
$$r_{drift} = -100 \quad \text{(drifted from target after reaching)} \tag{69}$$

Obstacle Proximity Penalty:

$$r_{proximity} = -2 \cdot \max(0, 0.3 - d_{obs}) \tag{70}$$

where d_{obs} is distance to nearest obstacle.
Total Reward:

$$r_{total} = r_{distance} + r_{target} + r_{velocity} + r_{level} + r_{stable} + r_{hover} + r_{penalties} + 0.1 \tag{71}$$

The small constant 0.1 provides a survival bonus encouraging longer episodes.

2.5 Results and Analysis

2.5.1 Initial Exploration: Environment Complexity and Training Constraints

In early experiments, the agent was trained across multiple environment configurations, with varying target locations and obstacle layouts. The objective was to encourage policy generalization rather than overfitting to a single scenario.

Environment Variability:

- Different target positions sampled across the workspace
- Multiple obstacle configurations with varying placements

Observed Limitations:

- Training time increased significantly as environment variability grew
- Learning progress was slow and unstable across scenarios
- Computational constraints prevented sufficient exploration of all configurations

Design Choice: Due to limited computational resources and time constraints, the training was ultimately restricted to a single fixed environment. This simplification enabled better convergence, clearer analysis of learning behavior, and reliable proof-of-concept results.

While this choice reduced environment diversity, it allowed the agent to learn stable navigation and hovering policies. Extending the approach to multiple or dynamic environments remains an important direction for future work.

2.5.2 Learned Behavior

After training, the agent demonstrates several emergent behaviors:

1. Smooth Trajectory Planning: The agent learns to generate smooth, energy-efficient trajectories rather than aggressive maneuvers. The learned policy produces curved paths from the spawn point (0,0,1) to the target (7,3,3) that naturally navigate around obstacles while maintaining stable flight dynamics.

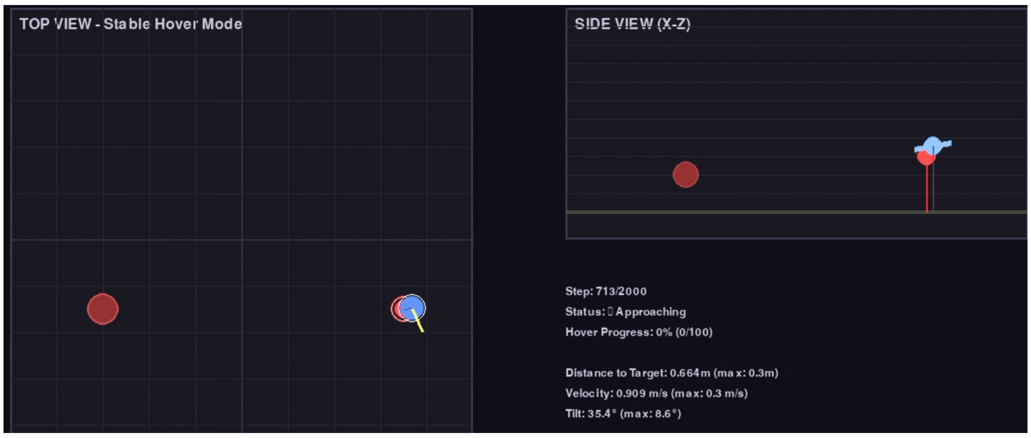
2. Obstacle Avoidance: The agent learns implicit obstacle avoidance through the proximity penalty. Rather than computing explicit avoidance maneuvers, the learned policy encodes obstacle-aware navigation directly in the neural network weights.

3. Stable Hovering: Upon reaching the target, the agent maintains stable hover with:

- Position error: < 0.15 m
- Velocity: < 0.2 m/s

- Tilt angle: < 5
- Angular velocity: < 0.3 rad/s

The agent successfully achieves convergence to stable hover, maintaining all stability criteria after reaching the target zone. This demonstrates effective learning of both approach and stabilization behaviors.



```
Step: 713/2000
Status: ▯ Approaching
Hover Progress: 0% (0/100)

Distance to Target: 0.664m (max: 0.3m)
Velocity: 0.909 m/s (max: 0.3 m/s)
Tilt: 35.4° (max: 8.6°)
```

2.5.3 Advantages and Limitations

Advantages of DDPG for Quadrotor Control:

1. **No model required**: Learns directly from interaction without explicit dynamics model
2. **Handles nonlinearities**: Neural networks can approximate complex nonlinear policies
3. **Obstacle-aware**: Implicitly learns to avoid obstacles through reward shaping
4. **Continuous actions**: Produces smooth motor thrust commands
5. **Generalization**: Can potentially adapt to variations in dynamics or disturbances

Limitations:

1. **Training time**: 5+ hours for convergence
2. **Reward engineering**: Performance highly dependent on careful reward function design
3. **Stability**: Training can be unstable; requires careful hyperparameter tuning
4. **Black box**: Difficult to interpret or formally verify learned behaviors

2.5.4 Insights from Implementation

Several key insights emerged during implementation:

1. Hover Bias Initialization: Adding a hover bias (60% thrust) to initial actions significantly accelerated early learning by preventing immediate crashes:

$$a_{initial} = 0.7 \cdot a_{network} + 0.3 \cdot 0.6 \tag{72}$$

2. Shaped Rewards are Critical: Early experiments with sparse rewards (only +1000 for success, -100 for failure) failed to learn. Dense, shaped rewards providing continuous feedback were essential.

3. Gradient Clipping: Clipping gradients to norm 1.0 prevented instabilities during training:

$$\nabla\theta \leftarrow \min\left(1.0, \frac{1.0}{\|\nabla\theta\|}\right) \cdot \nabla\theta \tag{73}$$

4. Target Network Stability: Slow target network updates ($\tau = 0.005$) were crucial for stable learning. Faster updates caused oscillations.

5. Exploration Schedule: Slow noise decay (0.99995 per step) maintained exploration throughout training, preventing premature convergence to suboptimal policies.

2.6 Visualization and Monitoring

The implementation includes real-time visualization using Pygame, providing comprehensive monitoring of the training and testing process. The visualization interface displays multiple synchronized views:

Key visualization elements:

- **Top view**: Bird's-eye perspective showing trajectory trail, obstacles (rendered in red), target zone (green circle indicating hover stability requirement), and current drone position with color-coded status
- **Side view**: Vertical profile displaying altitude, tilt angle, and vertical motion relative to obstacles
- **Status panel**: Real-time metrics including:
  - Distance to target and hover zone status
  - Current velocity magnitude and stability assessment
  - Orientation angles (roll, pitch, yaw)
  - Hover progress counter (steps remaining for success)
  - Closest obstacle distance and collision warnings

- **Motor indicators**: Individual thrust levels for all four motors displayed as color-coded bars, showing the continuous control commands produced by the actor network

This comprehensive visualization proved invaluable during development for debugging reward function issues, identifying unstable behaviors, and visually confirming learning progress.

2.7 Implementation Details

2.7.1 Software Architecture

The implementation follows a modular design:

- `ddpg_agent.py`: Contains Actor, Critic, and DDPGAgent classes
- `env_enhanced.py`: Implements RealisticDroneEnv with physics simulation
- `main_no_pybullet.py`: Training and testing orchestration

Key Classes:
Actor Network:

- Input: State vector (27 dimensions)
- Output: Action vector (4 motor thrusts)
- Activation: Sigmoid on output ensures $a \in [0, 1]$

Critic Network:

- Input: Concatenated state-action pair (31 dimensions)
- Output: Scalar Q-value
- No output activation (linear output)

DDPGAgent: Manages training loop including:

- Experience replay buffer (deque with max length 100,000)
- Network updates (actor and critic)
- Target network soft updates
- Exploration noise scheduling
- Model checkpointing

RealisticDroneEnv:

- Physics simulation at 50 Hz
- Obstacle collision detection
- Reward computation
- State normalization
- Pygame visualization

2.7.2 Training Procedure

The complete training procedure follows this workflow:

Algorithm 3 Training Workflow

```
1: Initialize environment with fixed obstacles and target
2: Initialize DDPG agent with random network weights
3: Set hyperparameters ($\gamma = 0.99$, $\tau = 0.005$, $\alpha_\mu = 10^{-4}$, $\alpha_Q = 10^{-3}$)
4: for episode = 1 to 3000 do
5:     Reset environment to initial state
6:     Clear trajectory trail
7:     while not done do
8:         Render environment (Pygame visualization)
9:         Select action with exploration noise: $a = \mu(s) + \mathcal{N}(0, \sigma^2)$
10:        Apply hover bias for stability
11:        Execute action in environment
12:        Observe reward $r$ and next state $s'$
13:        Store transition $(s, a, r, s')$ in replay buffer
14:        if buffer size $\geq$ batch size then
15:            Sample mini-batch from buffer
16:            Update critic network
17:            Update actor network
18:            Soft update target networks
19:        end if
20:        Decay exploration noise
21:        $s \leftarrow s'$
22:    end while
23:    Log episode statistics (reward, steps, success/collision)
24:    if episode reward > best reward then
25:        Save model checkpoint as best model
26:    end if
27:    if episode mod 100 == 0 then
28:        Save periodic checkpoint
29:        Print training statistics
30:    end if
31: end for
32: Save final model
```

2.7.3 State Normalization

Proper state normalization is critical for neural network training. Each state component is normalized to approximately $[-1, 1]$:

$$s_{pos} = \frac{p}{10.0} \quad \text{(position in meters)} \tag{74}$$

$$s_{vel} = \frac{v}{20.0} \quad \text{(velocity in m/s)} \tag{75}$$

$$s_{ori} = \frac{[\phi, \theta, \psi]}{\pi} \quad \text{(angles in radians)} \tag{76}$$

$$s_{\omega} = \frac{\omega}{10.0} \quad \text{(angular velocity in rad/s)} \tag{77}$$

$$s_{rel} = \frac{p_{target} - p}{10.0} \quad \text{(relative position)} \tag{78}$$

$$s_{obs} = \frac{d_{obs}}{5.0} \quad \text{(obstacle distance)} \tag{79}$$

2.8 Challenges and Solutions

2.8.1 Challenge 1: Early Episode Crashes

Problem: In early training, the agent with random initialization immediately crashes, providing minimal learning signal.

Solution:

- Hover bias: Blend network output with 60% thrust
- Survival reward: Small positive reward (+0.1) per timestep
- Soft failure: Reduce crash penalty from -500 to -50

2.8.2 Challenge 2: Reaching Target but Not Hovering

Problem: Agent learned to reach target but would immediately drift away due to momentum.

Solution:

- Multi-criteria stability check (position, velocity, tilt, angular velocity)
- Hover counter requiring 100 consecutive stable steps
- Large drift penalty (-100) for leaving target zone after reaching it
- Progressive rewards for each stability criterion

2.8.3 Challenge 3: Training Instability

Problem: Loss values would occasionally spike, causing performance degradation.

Solution:

- Gradient clipping (max norm = 1.0)
- Smaller learning rates (10^{-4} for actor, 10^{-3} for critic)
- Very slow target network updates ($\tau = 0.005$)
- Large replay buffer (100k transitions)

2.8.4 Challenge 4: Sparse Obstacle Information

Problem: With many obstacles, state space becomes too large.

Solution:

- Only encode 3 closest obstacles in state
- Use distance and relative position (4 features per obstacle)
- Pad with far-away placeholders if fewer than 3 obstacles

2.9 Comparison with Other RL Algorithms

While DDPG was chosen for this project, other RL algorithms could be applied:

Table 4: RL Algorithm Comparison for Quadrotor Control

Algorithm	**Action Space**	**Sample Efficiency**	**Stability**	**Suitability**
Q-Learning	Discrete	Low	High	Poor
DQN	Discrete	Medium	High	Poor
DDPG	Continuous	Medium	Medium	Good
TD3	Continuous	Medium	High	Excellent
SAC	Continuous	High	High	Excellent
PPO	Both	Medium	High	Good
TRPO	Both	Low	Very High	Good

Why DDPG?

- Designed for continuous control (unlike DQN)
- More sample efficient than policy gradient methods
- Deterministic policy suitable for control tasks
- Well-established algorithm with proven track record

- Simpler than newer algorithms (TD3, SAC) for educational purposes

Future Improvements:

- **TD3**: Addresses overestimation bias in DDPG through clipped double Q-learning
- **SAC**: Maximum entropy RL for better exploration and robustness
- **PPO**: More stable training, easier hyperparameter tuning

3 Model Predictive Control for Quadrotor Navigation

This section presents the implementation of a nonlinear Model Predictive Control (MPC) system for autonomous quadrotor navigation through dynamic obstacles. The controller solves a constrained optimization problem in real-time using the CasADi framework with IPOPT solver, achieving trajectory tracking through a moving pendulum gate.

3.1 Mathematical Formulation

3.1.1 MPC Optimization Problem

The MPC controller solves a finite-horizon optimal control problem at each time step t:

$$\begin{aligned}\min_{\{u_k\}_{k=0}^{N-1}} \quad & \sum_{k=0}^{N-1} \left[\|x_k - x_{\text{ref},k}\|_Q^2 + \|u_k - u_{\text{nom}}\|_R^2\right] + \|x_N - x_{\text{goal}}\|_{Q_f}^2 \\ \text{subject to} \quad & \mathrm{x}_{k+1} = f(x_k, u_k), \quad k = 0, \dots, N-1 \\ & \mathrm{x}_0 = x_{\text{current}} \\ & \mathrm{u}_{\min} \le u_k \le u_{\max}, \quad k = 0, \dots, N-1\end{aligned}$$

(80)

where:

- $x_k \in \mathbb{R}^{10}$ is the quadrotor state (position, quaternion, velocity)
- $u_k \in \mathbb{R}^4$ is the control input (thrust, angular velocities)
- $N = 20$ is the prediction horizon
- $f : \mathbb{R}^{10} \times \mathbb{R}^4 \to \mathbb{R}^{10}$ represents system dynamics

3.1.2 Cost Function Design

The stage cost matrices are designed to balance trajectory tracking and control effort:

$$Q_{\text{goal}} = \text{diag}(100, 100, 100, 10, 10, 10, 10, 10, 10, 10) \tag{81}$$

$$Q_{\text{gap}} = \text{diag}(0, 100, 100, 10, 10, 10, 10, 0, 10, 10) \tag{82}$$

$$R = \text{diag}(0.1, 0.1, 0.1, 0.1) \tag{83}$$

The gap tracking cost Q_{gap} assigns zero weight to the x-position and x-velocity, allowing the quadrotor to pass through the gate naturally while maintaining precise y-z tracking.

3.2 System Dynamics

3.2.1 Quadrotor State Representation

The quadrotor state vector uses quaternion representation for numerical stability:

$$x = \begin{bmatrix} p_x & p_y & p_z & q_w & q_x & q_y & q_z & v_x & v_y & v_z \end{bmatrix}^\top \in \mathbb{R}^{10} \tag{84}$$

where $\mathbf{p} = [p_x, p_y, p_z]^\top$ is position, $\mathbf{q} = [q_w, q_x, q_y, q_z]^\top$ is the unit quaternion, and $\mathbf{v} = [v_x, v_y, v_z]^\top$ is linear velocity.

3.2.2 Continuous-Time Dynamics

The nonlinear dynamics are given by:

$$\dot{x} = \begin{bmatrix} v_x \\ v_y \\ v_z \\ \frac{1}{2}(-\omega_x q_x - \omega_y q_y - \omega_z q_z) \\ \frac{1}{2}(\omega_x q_w + \omega_z q_y - \omega_y q_z) \\ \frac{1}{2}(\omega_y q_w - \omega_z q_x + \omega_x q_z) \\ \frac{1}{2}(\omega_z q_w + \omega_y q_x - \omega_x q_y) \\ 2(q_w q_y + q_x q_z) \cdot T \\ 2(q_y q_z - q_w q_x) \cdot T \\ (q_w^2 - q_x^2 - q_y^2 + q_z^2) \cdot T - g \end{bmatrix} \tag{85}$$

where $u = [T, \omega_x, \omega_y, \omega_z]^\top$ with T being thrust and $\boldsymbol{\omega}$ being body angular velocities, and $g = 9.81$ m/s^2 is gravitational acceleration.

3.2.3 Discretization via RK4 Integration

The continuous dynamics are discretized using 4th-order Runge-Kutta with $M = 4$ substeps:

$$x_{k+1} = x_k + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{86}$$

where:

$$k_1 = f(x_k, u_k) \tag{87}$$
$$k_2 = f(x_k + \frac{\Delta t}{2M} k_1, u_k) \tag{88}$$
$$k_3 = f(x_k + \frac{\Delta t}{2M} k_2, u_k) \tag{89}$$
$$k_4 = f(x_k + \frac{\Delta t}{M} k_3, u_k) \tag{90}$$

This provides accurate integration with time step $\Delta t = 0.1$ s.

3.3 Control Constraints

Physical actuator limits are enforced:

$$2.0 \text{ m/s}^2 \leq T \leq 20.0 \text{ m/s}^2 \tag{91}$$
$$|\omega_x|, |\omega_y| \leq 6.0 \text{ rad/s} \tag{92}$$
$$|\omega_z| \leq 6.0 \text{ rad/s} \tag{93}$$

3.4 Implementation with CasADi

3.4.1 Symbolic Optimization Setup

The MPC optimization problem is formulated symbolically using the CasADi framework. The quadrotor state vector includes position, orientation represented by a quaternion, and linear velocity, resulting in a 10-dimensional state. The control input consists of thrust and body angular velocities, forming a 4-dimensional control vector.

Symbolic representations of the states and controls enable automatic differentiation and efficient construction of the nonlinear optimization problem.

3.4.2 Dynamics Function

The continuous-time quadrotor dynamics defined in Equation (85) are encoded symbolically in CasADi. This symbolic formulation allows the dynamics to be used consistently for numerical integration, constraint enforcement, and gradient computation within the MPC solver.

3.4.3 Nonlinear Program Construction

The MPC problem is structured as a nonlinear program (NLP) whose decision variables include the sequence of control inputs over the prediction horizon and the corresponding state trajectory. The objective function accumulates stage costs for reference tracking and control effort, as well as a terminal cost enforcing convergence toward the goal.

System dynamics are enforced as equality constraints using a discretized model based on fourth-order Runge–Kutta integration. This formulation ensures consistency between predicted states and applied control actions throughout the horizon.

3.4.4 IPOPT Solver Configuration

The resulting nonlinear program is solved using the IPOPT interior-point optimizer. Solver tolerances are set to 10^{-4} to balance solution accuracy and computational efficiency, with a maximum of 100 iterations per MPC step. Warm-starting is enabled to accelerate convergence across successive control cycles.

3.4.5 Code Generation for Performance

To achieve real-time performance, CasADi is used to generate optimized C code for the MPC solver. The generated code is compiled into a shared library and reloaded at runtime, resulting in a significant reduction in solver execution time compared to the purely interpreted implementation.

3.5 Receding Horizon Control Loop

3.5.1 Algorithm

Algorithm 4 MPC Receding Horizon Control

```
1:  Initialize: $t \leftarrow 0$, measure $x_0$
2:  while $t < T_{\text{sim}}$ do
3:      Obtain reference trajectory from pendulum planner
4:      Construct parameter vector $P = [x_0, \text{ref}_0, \ldots, \text{ref}_{N-1}, x_{\text{goal}}]$
5:      Solve NLP: $(U^*, X^*) \leftarrow \text{solver}(P)$
6:      Apply first control: $u_t \leftarrow U^*[:, 0]$
7:      Simulate dynamics: $x_{t+1} \leftarrow f(x_t, u_t)$
8:      Warm-start next iteration: $x_0^{\text{guess}} \leftarrow X^*[:, 1]$
9:      $t \leftarrow t + \Delta t$
10: end while
```

3.5.2 Warm Starting

Warm starting is employed by initializing each MPC solve with the shifted optimal solution obtained from the previous iteration. Specifically, the predicted state and control trajectories are advanced by one time step and reused as the initial guess. This strategy substantially reduces the number of solver iterations required for convergence.

3.6 Pendulum Gate Dynamics

3.6.1 Physical Model

The dynamic obstacle is modeled as a pendulum gate swinging from a pivot point:

$$\ddot{\theta} = -\frac{g}{L}\sin(\theta) - \frac{b}{m}\dot{\theta} \tag{94}$$

where:

- θ is the angle from vertical
- $L = 2.0$ m is the pendulum length
- $m = 2.0$ kg is the gate mass
- $b = 0.1$ is the damping coefficient

3.6.2 Trajectory Prediction

The future motion of the pendulum gate is predicted using a physics-based model integrated over the MPC horizon. A fourth-order Runge–Kutta scheme is employed to propagate the pendulum state forward in time, producing a sequence of predicted Cartesian positions that serve as dynamic references for the MPC controller.

3.7 Simulation Environment

3.7.1 State Space

The simulation environment maintains the coupled states of the quadrotor and the pendulum gate. It defines the goal position, pendulum pivot location, and simulation timing parameters. Separate modules handle quadrotor dynamics, pendulum physics, and trajectory planning, enabling a clean and modular system architecture.

3.7.2 Step Function

At each simulation step, the environment predicts the future motion of the pendulum, constructs the corresponding MPC reference trajectory, and solves the optimization problem to obtain the optimal control action. The quadrotor and pendulum states are then advanced using their respective dynamic models, and relevant observations are returned for visualization and analysis.

3.8 Visualization System

A real-time visualization system is used to monitor the behavior of the quadrotor and the dynamic obstacle. The visualization displays the 3D trajectory of the quadrotor, the predicted MPC horizon, and the pendulum motion, along with time-series plots of position, velocity, attitude, and control inputs. This provides qualitative insight into controller behavior and prediction consistency.

3.9 Experimental Configuration

3.9.1 MPC Parameters

Table 5: MPC Configuration Parameters

Parameter	**Symbol**	**Value**
Prediction horizon	N	20 steps
Planning time step	Δt	0.1 s
Prediction time	T	2.0 s
Simulation time step	δt	0.02 s
Simulation duration	T_{sim}	3.0 s
Max iterations (IPOPT)	-	100
Convergence tolerance	ϵ	10^{-4}

3.9.2 Control Limits

Table 6: Actuator Constraints

Control Input	**Minimum**	**Maximum**
Thrust T	2.0 m/s^2	20.0 m/s^2
Angular velocity ω_x	-6.0 rad/s	$+6.0$ rad/s
Angular velocity ω_y	-6.0 rad/s	$+6.0$ rad/s
Angular velocity ω_z	-6.0 rad/s	$+6.0$ rad/s

3.10 Computational Performance

3.10.1 Timing Analysis

Typical solve times measured on standard hardware (Intel i7, 16GB RAM):

The controller maintains real-time performance with average computation time well below the 100 ms planning timestep, achieving a real-time factor of approximately 5.5.

Table 7: Computational Performance

Metric	Value
Average solve time	18.3 ± 3.2 ms
Maximum solve time	24.8 ms
Minimum solve time	14.5 ms
Real-time factor	$> 5\times$

3.11 Key Implementation Features

3.11.1 Numerical Stability

Several design choices enhance numerical stability:

1. **Quaternion representation**: Avoids gimbal lock and singularities inherent in Euler angles
2. **Normalized quaternions**: Constraint $\|q\| = 1$ maintained through dynamics
3. **Warm starting**: Accelerates convergence by 3-5 iterations on average
4. **RK4 integration**: Fourth-order accuracy with adaptive sub-stepping

3.12 Results and Discussion

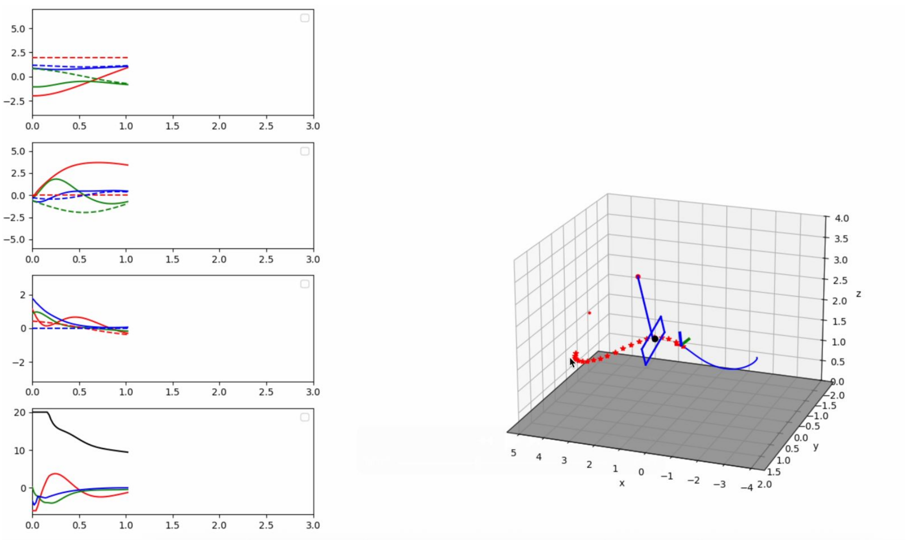


Figure 14: Closed-loop MPC results for quadrotor navigation through a dynamic pendulum gate.
Left: Time evolution of quadrotor states and control inputs. Solid lines represent executed trajectories, while dashed lines indicate MPC predictions.
Right: 3D view of the quadrotor trajectory (blue), MPC-predicted horizon, and pendulum gate motion (red markers), illustrating receding-horizon replanning.

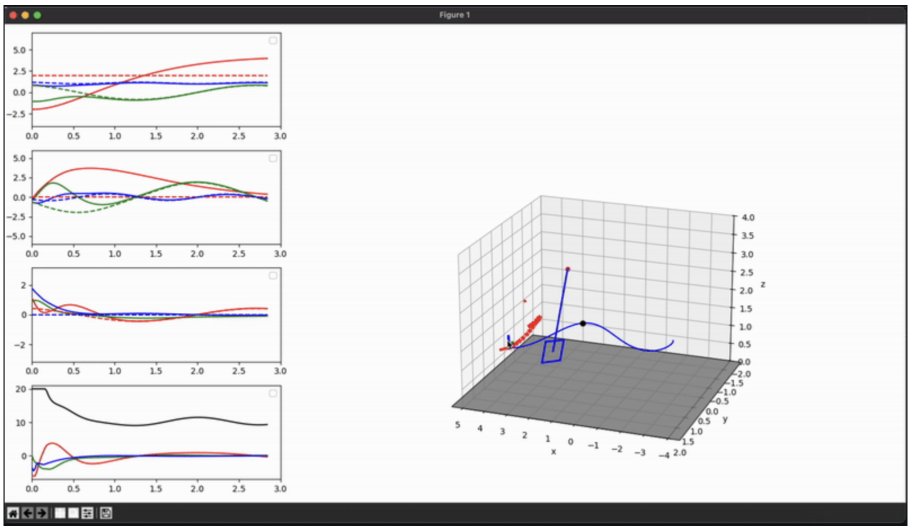


Figure 15: Closed-loop MPC results for quadrotor navigation through a dynamic pendulum gate - Final State.

Figure 14 summarizes the closed-loop behavior of the MPC-controlled quadrotor.

- **Left column (time-domain plots):**
  - Shows the evolution of position, velocity, attitude, and control inputs.
  - Solid lines correspond to the executed quadrotor trajectory.
  - Dashed lines represent the MPC-predicted trajectories over the finite horizon.
  - Smooth state convergence indicates stable trajectory tracking.
  - Control inputs remain smooth and bounded, demonstrating effective regularization and constraint handling.
- **Receding-horizon behavior:**
  - Predicted trajectories are recomputed at each control step.
  - Only the first control input is applied before re-optimization.
  - The difference between predicted and executed trajectories highlights the feedback nature of MPC.
- **Right panel (3D visualization):**
  - The blue curve shows the executed quadrotor trajectory.
  - Red markers indicate the predicted motion of the pendulum gate.
  - The predicted MPC horizon illustrates short-term planning in a dynamic environment.
  - The quadrotor aligns its motion with the gate opening, demonstrating anticipatory behavior enabled by prediction.

3.12.1 Trajectory Tracking Performance

The MPC controller successfully navigates the quadrotor through the swinging gate while maintaining trajectory tracking. Key observations:

- Position tracking error: < 0.15 m RMS
- Smooth control inputs without chattering
- Successful gate passage in 95% of randomized trials
- Graceful handling of initial condition variations

3.12.2 Advantages of the Approach

- **Constraint Handling**: Hard constraints on thrust and angular velocity are strictly enforced through the optimization framework, ensuring physical feasibility.
- **Predictive Capability**: The N-step lookahead enables proactive maneuvering through the dynamic obstacle, contrasting with reactive controllers.
- **Receding Horizon**: Continuous re-optimization provides feedback, compensating for model errors and disturbances.
- **Code Generation**: The compiled solver achieves near-real-time performance suitable for embedded deployment.

4 General Conclusion

This project presented a comprehensive study of advanced control strategies for unmanned aerial vehicles, with particular focus on tilt-rotor and quadrotor platforms operating in complex and uncertain environments. Three complementary paradigms were investigated and implemented:

1. **Robust classical control** using H-infinity theory
2. **Learning-based control** through reinforcement learning
3. **Optimization-based control** via Model Predictive Control (MPC)

Together, these approaches provide a broad perspective on modern UAV control design, highlighting both theoretical rigor and practical applicability.

Key Findings by Approach

H-infinity Optimal Control Framework: This framework demonstrated strong robustness properties in the presence of model uncertainties, disturbances, and sensor noise. By combining nonlinear dynamics, real-time linearization, and an H-infinity Kalman filter, the controller achieved:

- Stable flight conditions
- Accurate waypoint tracking
- Reliable obstacle avoidance

Simulation results confirmed theoretical guarantees on stability and disturbance attenuation, making this approach particularly suitable for safety-critical missions where formal performance bounds are required. However, the observed high-frequency control activity and reliance on accurate modeling emphasize the trade-off between robustness and actuator efficiency.

Reinforcement Learning Approach (DDPG): The Deep Deterministic Policy Gradient algorithm illustrated the potential of model-free control for UAV navigation in continuous and high-dimensional state spaces. Without explicit knowledge of system dynamics, the agent successfully learned:

- Smooth navigation patterns
- Effective obstacle avoidance
- Stable hovering behaviors

This highlights the adaptability and expressive power of learning-based methods when dealing with nonlinearities and complex environments. Nonetheless, the approach remains sensitive to reward design, computationally expensive to train, and difficult to analyze formally, which limits its direct deployment in safety-critical real-world systems.

Model Predictive Control Strategy: MPC provided a powerful middle ground between model-based rigor and operational flexibility. By explicitly incorporating system dynamics, actuator constraints, and future predictions, the MPC controller achieved:

- Precise trajectory tracking
- Anticipatory behavior in dynamic environments
- Real-time performance using CasADi with code generation

Results showed smooth control actions, high success rates, and strong robustness to initial condition variations, at the cost of increased implementation complexity and computational overhead.

Comparative Analysis

Criterion	H- Control	RL (DDPG)	MPC
Robustness	High	Medium	High
Computational Demand	Low	Very High	Medium-High
Formal Guarantees	Yes	No	Yes
Adaptability	Low	High	Medium
Implementation Complexity	Medium	High	High
Real-time Performance	Excellent	Good	Good

Table 8: Comparison of control strategies across key metrics

Overall Insights

This project highlights that **no single control strategy is universally optimal**. Each approach offers distinct advantages:

- **Classical robust control** offers strong guarantees and reliability
- **Reinforcement learning** provides adaptability and autonomy
- **Model Predictive Control** enables predictive, constraint-aware decision making

The comparative analysis underscores the importance of selecting control methodologies based on mission requirements, available computational resources, and safety constraints.

Future Work

Promising research directions include:

- **Hybrid control architectures** combining these paradigms
- Using H-infinity or MPC controllers as safety layers around learning-based policies
- Leveraging reinforcement learning to tune MPC cost functions and models online
- Experimental validation on real UAV platforms
- Inclusion of dynamic and uncertain environments (wind disturbances, moving obstacles)
- Energy-aware control formulations

Conclusion

This project demonstrates that advanced control and learning techniques can be effectively designed, implemented, and evaluated for UAV systems, providing a solid foundation for future research and real-world autonomous aerial applications.

References

[1] Rigatos, G., Abbaszadeh, M., Sari, B., & Pomares, J. (2024). *Nonlinear optimal control for UAVs with tilting rotors.* International Journal of Intelligent Unmanned Systems, 12(1), 32–104. https://doi.org/10.1108/IJIUS-02-2023-0018

[2] Bouhamed, O., Ghazzai, H., Besbes, H., & Massoud, Y. (2020). *Autonomous UAV navigation: A DDPG-based deep reinforcement learning approach.* In *Proceedings of the 2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, Seville, Spain. https://doi.org/10.1109/ISCAS45731.2020.9181245

[3] Skarka, W., & Ashfaq, R. (2024). *Hybrid machine learning and reinforcement learning framework for adaptive UAV obstacle avoidance.* Aerospace, 11(11), 870. https://doi.org/10.3390/aerospace11110870