Design, Implementation, and Comparison of Quadcopter Trajectory Tracking Controllers (PID, LQ, LQI, MPC)

Dustin Hartnett

University of Maryland, College Park

Maryland Applied Graduate Engineering (MAGE)

Robotics Engineering

12 August 2025

Contents

Abstract

This project aims to compare the performance of four different controller types applied to a quadcopter unmanned aerial vehicle (UAV) for trajectory tracking. A six degree-of-freedom quadcopter state space model was derived from the system's equations of motion and used as the basis for testing the four controllers: Proportional-Integral-Derivative (PID), Linear-Quadratic Regulator (LQR), Linear-Quadratic Regulator with Integral Action (LQI), and Model Predictive Control (MPC). Given the complex and coupled nature of a six degree-of-freedom quadcopter model, a genetic algorithm for automated tuning and optimization of each controller was developed. This was followed by the performance evaluation of each controller against three flight paths: a 3-dimensional line trajectory, an upwards spiral trajectory, and a rose petal curve trajectory with added sinusoidal z-motion. Wind was modelled as a simulated external disturbance acting as a varying side-blowing gust to evaluate the controllers against external unmeasured disturbances. Computation time, both 3-dimensional and 1-dimensional position root-mean-square error (RMSE), and oscillatory behavior and curvature analysis for determining trajectory accuracy and smoothness were the quantitative performance metrics chosen to compare controller performance. Overall, the LQI controller performed the best with quick computation time, great positional accuracy, and smooth resultant trajectories. The LQ and MPC controllers performed well, but the LQ controller struggled with handling external disturbances, and the MPC controller had a longer computation time with slightly less accuracy. The PID controller struggled with oscillations that resulted in system divergence on long, complex trajectories. The controllers were created in MATLAB instead of using Simulink to more accurately simulate the type of coding that would be necessary for use on a flight controller.

Introduction

This project involves the state space model derivation of a quadcopter UAV and the design and implementation of four controller types for trajectory tracking problems. A full literature review was completed before implementation, which involved reviewing academic and technical papers to gather an in-depth understanding of the functionality and implementation of each controller.

PID is a classic form of control with a large body of research available discussing various implementations on quadcopters for state regulation and trajectory tracking. Although this project is focused on trajectory tracking, papers discussing state regulation, meaning stabilizing a system around a single set-point, still offer valuable insight into controller implementation. Bayisa, 2019 [1] and Sahrir et al., 2022 [2] discuss the design and Simulink implementation of PID controllers for state regulation of a non-linear six degree-of-freedom quadcopter model in response to step inputs. Safarov, 2023 [3] discusses design and implementation of a PID controller with a linearized quadcopter model. This implementation is done in MATLAB and demonstrates the effectiveness of a simple PID controller for state regulation. The PhD thesis of Mellinger, 2012 [4] discussed many aspects of quadcopter PID control. It mainly focuses on complex, multi-UAV control algorithms, but also briefly discusses both linear and nonlinear six degree-of-freedom quadcopter modeling. In addition, it also delves into state regulation and

trajectory tracking for single quadcopters utilizing PID controllers. Abdulkareem et al., 2022 [5] discuss control of a nonlinear six degree-of-freedom quadcopter model using a proportional-derivative (PD) controller. This controller was implemented in MATLAB and demonstrated effectiveness in both state regulation and trajectory tracking despite the lack of an integral control component. Islam et al., 2017 [6] discuss the design and implementation of a PID controller, as well as a LQ controller, for control of a nonlinear six degree-of-freedom quadcopter for trajectory tracking. Their results display that even though a PID controller is effective, an LQR controller can be more effective and easier to implement.

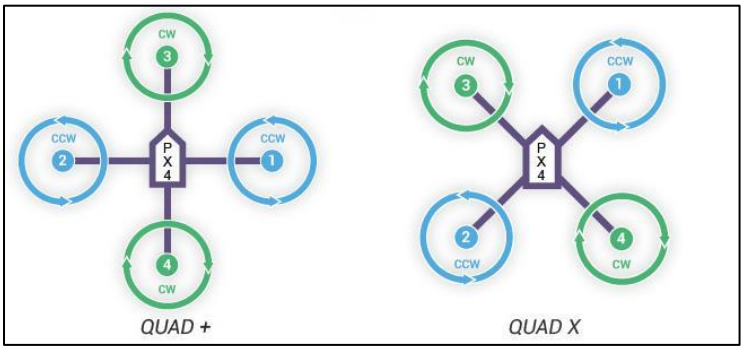
Although classical techniques like PID controllers are historically popular for quadcopter design, increasingly LQR, LQI, and MPC controllers are being researched and developed in academia and industry. Velagic et al., 2022 [7] compare the performance of PID and LQR controllers when applied to a linear time invariant, six degree-of-freedom octocopter model. The authors implemented cascaded LQR controllers for attitude and orientation for trajectory tracking problems, highlighting the improved performance with LQR controllers compared to PID. Nekoo et al., 2024 [8] discuss the implementation of a LQR controller on a linear time variant, bird-like robot for trajectory tracking. Although not directly applicable to this project, the authors discuss the concept of backwards integration of the differential Ricatti equation for gain scheduling with a linear time variant system, an interesting technique that would be essential if this work focused on a linear time variant quadcopter model. Martins et al., 2019 [9] discuss the implementation of a LQI controller on a linear time invariant, six degree-of-freedom quadcopter model for trajectory tracking problems. They evaluated the controllers on a real UAV to compare experimental data to simulation data, and found the controller performed as well as they expected.

Ganga et al., 2017 [10] discuss the comparison of a PID controller and a linear MPC controller implemented on a six degree-of-freedom quadcopter model for state regulation. They discuss the design of the controllers as well as the benefits of using MPC controllers over PID controllers for coupled systems. Abougarair et al., 2025 [11] compare a linear MPC controller and an LQR controller implemented on a six degree-of-freedom quadcopter model for trajectory tracking. They highlight the excellent performance of both controllers but emphasize the general robustness and versatility of the MPC compared to the LQR. Islam et al., 2017 [12] discuss the implementation of a linear MPC controller on a six degree-of-freedom quadcopter model for trajectory tracking and highlight the controller's success at rejecting unmeasured noise and disturbances. Finally, Varma et al., 2020 [13] compare PID, LQR and MPC controllers for trajectory tracking with a car model. Although not a quadcopter model, the paper highlights the implementation and performance differences of each controller which made it a great foundational piece before beginning implementation on this project.

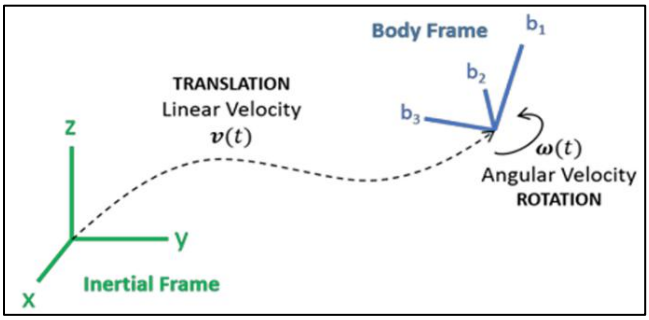
In this project, information from all aspects of the literature review was leveraged to design and implement the four controller types on the linearized quadcopter model. The four controller types each posed unique challenges and demonstrated clear advantages and disadvantages with regards to trajectory tracking capabilities. The first step in this project was the derivation of the linearized state space model for a six degree-of-freedom quadcopter UAV.

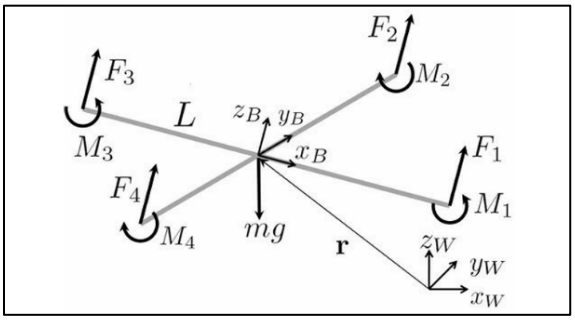
System Modeling

For this project, quadcopter UAV was designed as a “quad-plus” configured quadcopter – a configuration resembling a plus-sign as opposed to an X. In this configuration, two motors are responsible for roll torque and two for pitch torque, as opposed to control mixing of all four motors for each directional torque. [15]



A full quadcopter model involves a six degree-of-freedom system with three translation degrees-of-freedom - x, y, and z - and three rotational degrees-of-freedom - roll, pitch, and yaw. For this project, the x-direction is forward-facing, y-direction left-facing, and z-direction upwards-facing. Roll is rotation around the x-axis, pitch is rotation around the y-axis, and yaw is rotation around the z-axis. A common practice in UAV modeling is to model your desired trajectory with respect to the world frame but calculate your system states and control inputs with respect to the body frame. [4] [11]





Another common practice is to model the quadcopter as an underactuated system with twelve system states – six translation and rotational states and six derivatives – and four control inputs. For simulation simplicity, the quadcopter model inputs are often modeled as a total thrust value, a roll torque, a pitch torque, and a yaw torque. The controller in this project outputs the four control inputs for the system – total thrust and three torques – which can be converted to desired propeller angular velocities and scaled to desired pulse width modulation values for implementation on a physical system as follows [5]:

$$\omega_{1,2,3,4} = \text{Propeller angular velocities} \quad k_t = \text{Propeller coefficient of thrust}$$
$$k_b = \text{Propeller coefficient of drag} \quad l = \text{Distance from center of mass to propeller}$$

$$U_1 = \text{Total Thrust} \quad U_2 = \text{Roll Torque} \quad U_3 = \text{Pitch Torque} \quad U_3 = \text{Yaw Torque}$$

$$\begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix} = \begin{bmatrix} k_t & k_t & k_t & k_t \\ 0 & -lk_t & 0 & lk_t \\ -lk_t & 0 & lk_t & 0 \\ -k_b & k_b & -k_b & k_b \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix}$$

To convert translation and rotation from the world to body frame for calculating the system equations of motion, a 3-dimensional rotation matrix, *R*, is used which is comprised of rotation matrices for x, y, and z rotation.

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & sin(\phi) & cos(\phi) \end{bmatrix} \quad R_y = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix} \quad R_z = \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R = R_z * R_y * R_x$$

$$R = \begin{bmatrix} cos(\theta)cos(\psi) & sin(\phi)sin(\theta)cos(\psi) - cos(\phi)sin(\psi) & cos(\phi)sin(\theta)cos(\psi) + sin(\phi)sin(\psi) \\ cos(\theta)sin(\psi) & sin(\phi)sin(\theta)sin(\psi) + cos(\phi)cos(\psi) & cos(\phi)sin(\theta)sin(\psi) - sin(\phi)cos(\psi) \\ -sin(\theta) & sin(\phi)cos(\theta) & cos(\phi)cos(\theta) \end{bmatrix}$$

To derive the equations of motion for the transitional states of the quadcopter, forces were modeled in the world frame and the rotation matrix was applied to convert them to the body frame. The nonlinear translational acceleration can be modeled as follows:

$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R \begin{bmatrix} 0 \\ 0 \\ U_1 \end{bmatrix}$$

$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \frac{1}{m} \begin{bmatrix} U_1[sin(\phi)sin(\psi) + cos(\phi)sin(\theta)cos(\psi)] \\ U_1[-sin(\phi)cos(\psi) + cos(\phi)sin(\theta)sin(\psi)] \\ U_1[cos(\phi)cos(\theta)] - g \end{bmatrix}$$

For rotational states, the angular velocities of the quadcopter in the body frame are often designated as *p, q,* and *r*. Equations for the body angular accelerations with respect to the angular velocities and applied control torques were derived as follows:

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}, \quad \omega = \begin{bmatrix} p \\ q \\ r \end{bmatrix}, \quad \dot{\omega} = \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix}, \quad \tau = \begin{bmatrix} U_2 \\ U_3 \\ U_4 \end{bmatrix}$$

$$I\dot{\omega} = -\omega \times I\omega + \tau$$

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} \frac{1}{I_{xx}}[(I_{yy} - I_{zz})qr + U_2] \\ \frac{1}{I_{yy}}[(I_{zz} - I_{xx})pr + U_3] \\ \frac{1}{I_{zz}}[(I_{xx} - I_{yy})pq + U_4] \end{bmatrix}$$

The body angular velocities are related to Euler angular velocities and are derived from a combination of rotation matrices and Euler angular velocities to convert body angular motion to Euler angular motion as follows [11]:

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \dot{\psi}R_zR_y + \dot{\theta}R_z + \dot{\phi}$$

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} 1 & 0 & -sin(\theta) \\ 0 & cos(\phi) & sin(\phi)cos(\theta) \\ 0 & -sin(\phi) & cos(\phi)cos(\theta) \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}$$

A common practice when modeling a linearized quadcopter model performing controlled maneuvers, as done in this project, is to assume linearity along a trajectory, a constant yaw angle of zero degrees, and a system state close to hover. This allows us to assume that *p, q,* and *r* are equivalent to the Euler angular velocities due to the quadcopter state not straying too far from forward-facing hover along its trajectory. Therefore, the resultant equations for rotational motion are as follows [4]:

$$\dot{\phi} = p, \quad \dot{\theta} = q, \quad \dot{\psi} = r$$

$$\begin{bmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} \frac{1}{I_{xx}}[(I_{yy} - I_{zz})\dot{\theta}\dot{\psi} + U_2] \\ \frac{1}{I_{yy}}[(I_{zz} - I_{xx})\dot{\phi}\dot{\psi} + U_3] \\ \frac{1}{I_{zz}}[(I_{xx} - I_{yy})\dot{\phi}\dot{\theta} + U_4] \end{bmatrix}$$

Due to the assumption that the UAV stays near a hover state for the duration of its trajectory, linearization around a trajectory with Jacobian methods is not necessary and small-angle and hover-state approximations can be used. Gain scheduling for different sections of flight is also not a necessary step for a quadcopter performing smooth, controlled flight. For this project, the trajectories were designed with a constant yaw value of 0 degrees. This assumption,

combined with small angle assumptions and assumptions of a constant state near hover allows the linearization of the system as follows [4]:

$$\phi = \theta = \psi = 0$$
$$\dot{\phi} = \dot{\theta} = \dot{\psi} = 0$$
$$cos(\phi) = cos(\theta) = cos(\psi) = 1$$
$$sin(\phi) = \phi$$
$$sin(\theta) = \theta$$
$$sin(\psi) = 0$$

After system linearization, the resultant state space equation can be written with assumptions for full state observability given a realistic implementation of altimeters, IMUs, and GPS units as follows:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & mg & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -mg & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{m} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \frac{1}{I_{xx}} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{I_{yy}} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{I_{zz}} \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \quad D = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad X = \begin{bmatrix} x \\ \dot{x} \\ y \\ \dot{y} \\ z \\ \dot{z} \\ \phi \\ \dot{\phi} \\ \theta \\ \dot{\theta} \\ \psi \\ \dot{\psi} \end{bmatrix}, \quad \dot{X} = \begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dot{y} \\ \ddot{y} \\ \dot{z} \\ \ddot{z} \\ \dot{\phi} \\ \ddot{\phi} \\ \dot{\theta} \\ \ddot{\theta} \\ \dot{\psi} \\ \ddot{\psi} \end{bmatrix}$$

$$\dot{X} = AX + BU, \quad Y = CX + DU$$

The system requires compensation for acceleration due to gravity in the z-direction that is unaccounted for in a linearized system and instead needs to be manually programmed into each controller after each state update to properly account for gravity's effects on the quadcopter states.

After linearization, it is important to check the system controllability before designing a controller around it. Controllability means the linearized system can be controlled from an initial system state to a desired final state if given the proper control inputs. In this case, it would mean the linearized system can be controlled and guided along a trajectory if the system stays near a hover state. Controllability is evaluated with the controllability matrix, with controllability being confirmed if the controllability matrix is of full rank. For this model with twelve states, the rank, *n*, needs to be twelve. In this case, the rank was confirmed to be twelve and the linearized quadcopter model was determined to be controllable.

$$\mathcal{C} = [B \quad AB \quad A^2B \quad \dots \quad A^{n-1}B]$$
$$rank(\mathcal{C}) = n$$

Operational limitations needed to be applied to the quadcopter to make the simulation comparable to real quadcopter capabilities. The thrust required for hover, *mg*, for a quadcopter that has a mass of 1.25 kg is about 12.25 N. Quadcopters designed for controlled, smooth flight, not racing or aggressive maneuvers, often have a maximum available thrust of two-times the hover thrust. For this project, the maximum total thrust was limited to 25 N. In addition, the roll, pitch, and yaw torques were limited to 7 Nm to ensure the controller would not introduce aggressive maneuvers that are beyond the capabilities of a standard quadcopter.

Controller Overviews and Algorithms

In this project, four unique controllers were designed and implemented: a PID controller, LQR controller, LQI controller, and MPC controller. Due to the linearization of the quadcopter model being time invariant around a hover point as it moves, gain scheduled controllers were not necessary and the control law could be designed before the simulation as opposed to concurrently with each time step.

The state model was derived in continuous time space; however, the controllers were implemented using a discrete time setup. The trajectories, state space model, and control law were designed prior to the simulation. During simulation, the system error was calculated, control input determined, and state updated with each time step from the initial state to final desired state. The state was updated based on the discrete state space model of the system:

$$\dot{X}_i = AX_i + BU_i$$
$$X_{i+1} = X_i + \dot{X}_i dt$$

After the system setup, controller design, and simulation setup, each controller was repackaged as a separate cost function and ran through a genetic algorithm tuning process to determine the best gains, costs, or weights for controller performance. More details on the

genetic algorithm tuning are provided in the Controller Tuning section. The following subsections describe each controller and the required setup for integration with a quadcopter model.

PID

Despite the PID controller being a historically popular pick for quadcopters and many other systems, it is not the most effective or easily implemented controller. At its simplest, a PID controller is a controller that leverages user-defined gains and manipulates state error, the integral of state error over time, and the derivative of the state error over time in a combined function to determine the desired control input for a system. The general form of a PID controller is as follows [5]:

$$r(t) = \text{Reference Trajectory}, \quad X(t) = \text{System State}$$
$$U(t) = \text{Control Input}, \quad e(t) = \text{State Error}$$
$$K_p = \text{Proportional Gain}, \quad K_i = \text{Integral Gain}, \quad K_d = \text{Derivative Gain}$$
$$e(t) = r(t) - X(t), \quad U(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{d}{dt} e(t)$$

The three terms – proportional, integral, and derivative – each play an important part in the control law. The proportional term contributes a component to the control input directly proportional to the state error, moving the system towards the desired state. The integral term contributes a component to the control input proportional to the integral of the state error over time. This helps the system account for persistent disturbances or offsets and long-term errors. The derivative term contributes a component to the control input proportional to the change in error over time. This accounts for the rate of change of error and helps the system avoid overshooting the desired state as the state errors approach zero.

A PID controller is a highly effective controller type for single-input-single-output (SISO) systems or uncoupled systems, where one input directly affects one system state. However, a quadcopter model is a multiple-input-multiple-output (MIMO), coupled model that requires control mixing. Additionally, unlike with the LQR, LQI, or MPC controllers where a desired roll and pitch angle of zero is acceptable, the PID controller requires a desired roll and pitch angle to be calculated with each iteration. This results in cascaded PID controllers: one set for determining the desired roll and pitch angles given positional state errors and one set that uses those desired angles, along with the desired yaw angle of zero degrees, to calculate the necessary control inputs to the system. Using the cascaded approach, the control inputs can technically be decoupled while still considering all six degrees-of-freedom for state adjustments.

The first step is determining the required total thrust the quadcopter needs at each time step. This is determined from manipulating the equations of motion, and accounts for z-direction error, for body rotation, and acceleration due to gravity.

$$m\ddot{z} = mg + U_1 cos(\phi)cos(\theta)$$
$$U_1 = \frac{m}{cos(\phi)cos(\theta)}[g + \ddot{z}]$$
$$\ddot{z} = \text{PID Control Law Contribution}$$
$$U_1 = \frac{m}{cos(\phi)cos(\theta)}[g + K_{p,z}e_z(t) + K_{i,z}\int_0^t e_z(\tau)d\tau + K_{d,z}\frac{d}{dt}e(t)]$$

From there, total thrust input and the quadcopter's positional error in the x- and y-directions is used to determine the necessary pitch and roll angles, respectively. The equations for the desired pitch and roll angles are derived by manipulating the equations of motions, and accounting for vectored thrust and positional error of the quadcopter.

$$m\ddot{x} = U_1 sin(\theta), \quad m\ddot{y} = U_1 sin(\phi)$$
$$\theta \cong \frac{1}{U_1}m\ddot{x}, \quad \phi \cong \frac{1}{U_1}m\ddot{y}$$
$$\ddot{x}, \ddot{y} = \text{Desired Acceleration} + \text{PID Control Law Contribution}$$
$$\theta_{des} = \frac{1}{U_1}[m\ddot{x}_{des} + K_{p,x}e_x(t) + K_{i,x}\int_0^t e_x(\tau)d\tau + K_{d,x}\frac{d}{dt}e_x(t)]$$
$$\phi_{des} = \frac{1}{U_1}[m\ddot{y}_{des} + K_{p,y}e_y(t) + K_{i,y}\int_0^t e_y(\tau)d\tau + K_{d,y}\frac{d}{dt}e_y(t)]$$

With the desired yaw angle of zero degrees being constant, the necessary control torques are calculated based on the error in angular states.

$$U_2 = K_{p,\phi}e_\phi(t) + K_{i,\phi}\int_0^t e_\phi(\tau)d\tau + K_{d,\phi}\frac{d}{dt}e_\phi(t)$$
$$U_3 = K_{p,\theta}e_\theta(t) + K_{i,\theta}\int_0^t e_\theta(\tau)d\tau + K_{d,\theta}\frac{d}{dt}e_\theta(t)$$
$$U_4 = K_{p,\psi}e_\psi(t) + K_{i,\psi}\int_0^t e_\psi(\tau)d\tau + K_{d,\psi}\frac{d}{dt}e_\psi(t)$$

Because decoupling the controller while maintaining a realistic system requires using nonlinear dynamics, the four control inputs are then input into the nonlinear equations of motion to determine the change in state. Linear and angular accelerations are first calculated. Then the linear accelerations are double integrated to calculate the updated linear velocities and positions, and the body angular accelerations are integrated to calculate the updated body angular velocities, which are then used to calculate the Euler angular velocities. The Euler angular velocities can then be integrated to determine the change in roll, pitch, and yaw angles. The new system state is saved and passed back to the controller for the next iteration.

The controller gains optimized by the genetic algorithm are as follows:

$K_{p,x}$	6.45	$K_{p,\phi}$	8.86
$K_{i,x}$	0.089	$K_{i,\phi}$	0.00004

$K_{d,x}$	0.00011	$K_{d,\phi}$	0.0004
$K_{p,y}$	7.34	$K_{p,\theta}$	8.43
$K_{i,y}$	0.00006	$K_{i,\theta}$	0.00062
$K_{d,y}$	0.51	$K_{d,\theta}$	0.00011
$K_{p,z}$	25	$K_{p,\varphi}$	3.51
$K_{i,z}$	0.97	$K_{i,\varphi}$	0.008
$K_{d,z}$	9.03	$K_{d,\varphi}$	0.0053

LQR

A LQR controller, or linear quadratic regulator (LQR), is a type of controller that leverages a quadratic cost function and a linear state space model for defining its control law. For trajectory tracking, the standard LQR setup needs to be modified to account for a desired state trajectory as well as a desired input trajectory. For this project, the desired linear state positions and velocities change over time and the angular state values remain a constant zero degrees and degrees per second. Assumed hover conditions allow this technique to be used and is common in literature. The desired reference controls were assumed to be zero at all time steps except for the total thrust, which is defined as *mg,* and is the total thrust required for hover conditions. After defining the reference trajectory and control inputs, the first step in designing an LQR controller is to define modified state and input values by accounting for state and input trajectories, and defining and minimizing the cost function, *J*, by selecting *Q* and *R* cost matrices that adequately penalize state error and aggressive control inputs, respectively.

$$X(t) = \text{System State}, \quad X_{des}(t) = \text{Desired State Trajectory}$$
$$U(t) = \text{Control Input}, \quad U_{des}(t) = \text{Desired Control Input}$$
$$\widetilde{X}(t) = X(t) - X_{des}(t), \quad \widetilde{U}(t) = U(t) - U_{des}(t)$$

The modified state and input matrices, along with the cost matrices *Q* and *R*, can be used to solve the optimal “cost-to-go” function, *J*. The Hamilton-Jacobi-Bellman (HJB) equation is a quadratic cost function that the LQR solution is designed to solve, and I select the solution such that it uses the matrix *P*.

$$HJB: 0 = \min_{U}[X^T Q X + U^T R U + \frac{\partial J^*}{\partial x}(AX + BU)]$$
$$Solution: J^* = X^T P X, \quad \frac{\partial J^*}{\partial x} = 2X^T P$$

Because the minimized equation is quadratic and convex, the equation can be minimized and simplified, finding the optimal value *U*, and hence *K,* given the definition of the control law that satisfies the conditions.

$$\frac{\partial}{\partial U} = 2U^T R + 2X^T PB = 0$$
$$U = -R^{-1}B^T PX = -KX$$
$$K = R^{-1}B^T P$$

Plugging this back into the HJB and simplifying the equation leaves me with the following equation that can be used to determine the optimal control gain matrix, *K,* given the solution to the HJB and algebraic Ricatti equation, *P*. [15]

$$A^T P + PA - PBR^{-1}B^T P + Q = 0$$

Because the quadcopter is assumed to be near a hover state at each time step, only one K gain matrix must be solved for, as opposed to gain scheduling with multiple K matrices defined at different time steps. The control law can be calculated and simplified as follows:

$$A^T P + PA - PBR^{-1}B^T P + Q = 0$$
$$K = R^{-1}B^T P$$
$$\widetilde{U}(t) = -K\widetilde{X}(t)$$
$$\widetilde{X}(t) = X(t) - X_{des}(t), \quad \widetilde{U}(t) = U(t) - U_{des}(t)$$
$$U(t) = U_{des}(t) - K(X(t) - X_{des}(t))$$

The control law is used at each time step to stabilize the system in discrete time. The controller leveraged MATLAB's built-in *lqr* function. The function requires the user to define the linearized state space model and the *Q* and *R* cost matrices. It outputs the gain matrix *K* which is used to define the control input based on state error and reference control inputs. The *Q* and *R* cost matrices optimized by the genetic algorithm are as follows [16]:

$$Q = \begin{bmatrix} 449 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 161 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 392 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 94.6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 486 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 58.2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 13.1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.91 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 310 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 297 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 135 \end{bmatrix}$$

$$R = \begin{bmatrix} 0.104 & 0 & 0 & 0 \\ 0 & 0.139 & 0 & 0 \\ 0 & 0 & 0.033 & 0 \\ 0 & 0 & 0 & 8.86 \end{bmatrix}$$

The LQR control law is a highly effective and computationally inexpensive control law to use for system stabilization and theoretically works very well for trajectory tracking when continuous near-hover conditions are assumed throughout flight. However, due to its lack of integral action for error tracking, it is not well suited for handling errors introduced by continuous, unmeasured disturbances, such as wind.

LQI

The LQI controller, or linear quadratic regulator with integral action (LQRI), uses the same basic structure as the LQR controller, but is augmented with additional states that account for integral errors. The additional augmented states help the controller account and correct for errors introduced by continuous, unmeasured disturbances, such as wind. For this project, chose to only include integral action was implemented for only the x, y, and z states of the quadcopter with the assumption that they should be sufficient for handling offsets along the trajectory. The augmented state space matrices are as follows:

$$X_I = \int_0^t \tilde{X}_{x,y,z}(t)dt, \quad X_I(0) = 0$$

$$\overline{X}(t) = \begin{bmatrix} \tilde{X} \\ X_I \end{bmatrix}, \quad \dot{\overline{X}}(t) = \begin{bmatrix} \dot{\tilde{X}} \\ \dot{X}_I \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\overline{A} = \begin{bmatrix} A & 0_{12\times 3} \\ -C & 0_{3\times 3} \end{bmatrix}, \quad \overline{B} = \begin{bmatrix} B \\ 0_{3\times 4} \end{bmatrix}$$

Additionally, the *Q* cost matrix and *K* gain matrix become augmented with three additional costs and gains for the three additional integrated state variables. This changes the cost function, gain matrix calculation, and control law derivation as follows:

$$\overline{Q} = \begin{bmatrix} Q & 0 \\ 0 & Q_I \end{bmatrix}$$

$$J = \int_{t_0}^{\infty} \overline{X}(t)^T \overline{QX}(t) + \tilde{U}(t)^T R\tilde{U}(t)dt$$

$$\overline{A}^T P + P\overline{A} - P\overline{B}R^{-1}\overline{B}^T P + \overline{Q} = 0$$

$$\overline{K} = R^{-1}\overline{B}^T P = [K \quad K_I]$$

$$\tilde{\overline{U}}(t) = -\overline{K}\tilde{\overline{X}}(t)$$

$$U(t) = U_{des}(t) - K(X(t) - X_{des}(t)) - K_I X_I$$

This control law is implemented at each time step in the simulation to stabilize the quadcopter model along its desired trajectory while also correcting for persisting errors or

disturbances. For the implementation of this controller, MATLAB's *lqi* function was leveraged, which outputs the augmented gain matrix if fed an augmented state space system, augmented Q cost matrix, and R cost matrix. The tuned cost matrices optimized by the genetic algorithm are as follows [17]:

$$\overline{Q} = \begin{bmatrix} 500 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.196 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 378 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 75.4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 500 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 238 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 169 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 94.6 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.015 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 125 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 234 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 500 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 443 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.002 \end{bmatrix}$$

$$R = \begin{bmatrix} 0.313 & 0 & 0 & 0 \\ 0 & 0.143 & 0 & 0 \\ 0 & 0 & 0.04 & 0 \\ 0 & 0 & 0 & 9.4 \end{bmatrix}$$

MPC

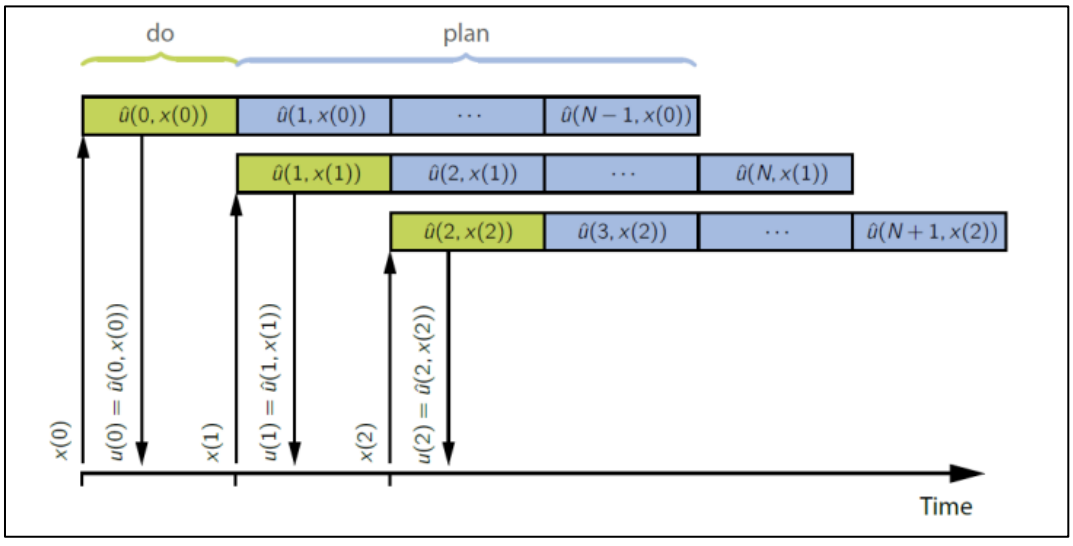
An MPC controller is a discrete-time controller that leverages an optimized cost function, like the LQR and LQI controllers, to determine the optimal control inputs for a receding finite horizon problem. For this project, MATLAB MPC toolbox and built-in *mpc* function was used which solves a quadratic programming optimization problem to determine the optimal control inputs over time. [18]

An MPC controller requires multiple parameters to define its setup and function. The prediction horizon is how far out the controller can "see", meaning at each time step the controller can predict the desired state of the system out to a user-defined horizon. The control horizon is how far out the controller will compute optimal control inputs, meaning it will compute a finite number of control inputs equal to the control horizon that will ensure the system closely tracks its desired trajectory. Based on literature, the control horizon is typically 20-30% of the prediction horizon. [10] These are key parameters as large horizon values will allow the controller to see and predict out to a far horizon but with high computational costs, while shorter prediction and control horizons will be less computationally expensive at the cost of the controller having limited "visibility". For this implementation, a prediction horizon of twelve steps

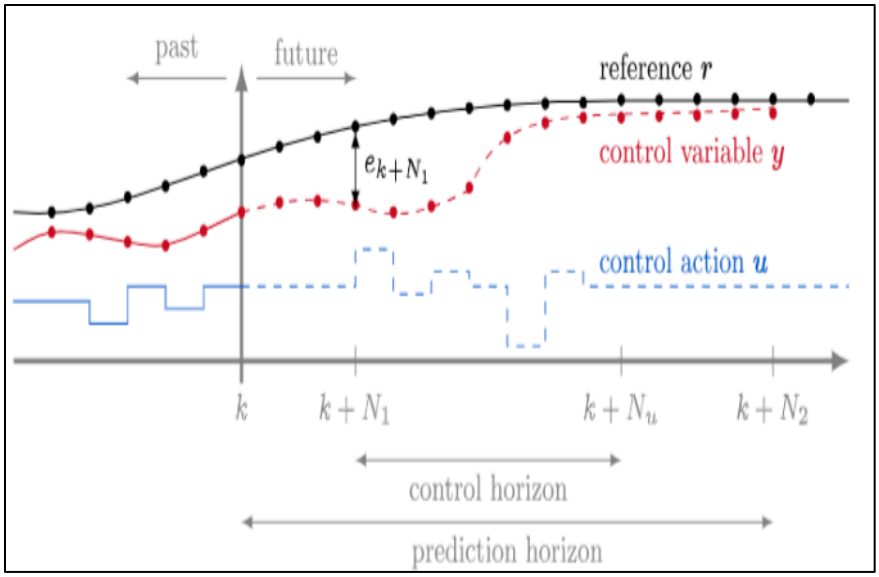
and the control horizon of four steps were selected. This combination resulted in a good balance of computational efficiency and trajectory tracking accuracy.

An MPC controller also allows for user-defined constraints on control inputs and states. These are fed to the controller as either hard or soft constraints, meaning the controller tries to stay within them or must stay within them, respectively. These constraints are helpful for controller design as they allow the user to explicitly define value constraints in the controller's optimization problem as opposed to integrating post-calculation constraints on the control input and state values that will result in non-optimized control inputs. For this project, hard control input constraints were implemented to be consistent with the other controllers, with thrust limited to 25 N and the torques limited to 7 Nm.

With each time step, the MPC controller "looks" out to the prediction horizon to see the desired reference trajectory and end state. It then takes the current state and the desired end state and computes the control inputs required to bring the system to the desired trajectory on its way to the end state. It sees as far as the prediction horizon and computes as many control inputs to bring the system towards the desired trajectory as the control horizon is defined. It then applies the first calculated control input and "remembers" the remaining calculated control inputs. At the next time step, it will repeat the process with updated state measurements and recompute the optimized control inputs out to the control horizon given the updated state, previously calculated control inputs as reference, and newly updated desired states as the prediction horizon shifts. [11] [14]



Example of the Receding Horizon Nature of MPC Control [11]



Example of Horizons and Control Input Calculation [14]

The MPC controller uses a quadratic cost function, *J*, to optimize the control inputs. It computes a combined cost function for however many control inputs are defined by the control horizon to guide the system towards its reference trajectory. The cost function accounts for user-defined state or control input constraints and defines the optimal set of sequential control inputs as the set that minimizes the overall cost function. [19]

$$h_c = \text{Control Horizon}$$
$$Q = \text{System State Weights - Penalize State Error}$$
$$R_U = \text{Control Input Weights - Penalize Aggressive Control Inputs}$$
$$R_{\Delta U} = \text{Control Input Rate Weights - Penalize Aggressive Control Input Changes}$$
$$X_{ref} = \text{Reference Trajectory}, \quad U_{ref} = \text{Reference Inputs}$$
$$k = \text{Current Control Interval}, \quad i = \text{Time Step}$$

$$e_X(i+k) = X_{ref}(k+i+1|k) - X(k+i+1|k)$$
$$e_U(i+k) = U_{ref}(k+i|k) - U(k+i|k)$$
$$\Delta U(k+i) = U(k+i|k) - U(k+i-1|k)$$
$$z_k = \text{Decision Variables} = [U(k|k)^T \quad U(k+1|k)^T \quad \dots \quad U(k+h_c-1|k)^T]^T$$

$$J(z_k) = \sum_{i=0}^{h_c-1} \left\{ [e_X^T(k+i) Q e_X(k+i)] + [e_U^T(k+i) R_U e_U(k+i)] + [\Delta U^T(k+i) R_{\Delta U} \Delta U(k+i)] \right\}$$

After optimization, the controller applies the first calculated control input from the optimal set of control inputs and then repeats the process at the next time-step. This process works very well for stabilizing a system around a trajectory in simple conditions. However, like the LQR controller, this version of the MPC controller is not optimized to handle persistent disturbances

and offsets, such as wind. Additional integral action terms, like in the LQI controller, are necessary for handling that. The tuned weights optimized by the genetic algorithm for state variables, control inputs, and control input rates are as follows:

$$Q = \begin{bmatrix} 42.15 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 200 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 147 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 115 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 200 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 22 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8.1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5.125 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 149 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 189 \end{bmatrix}$$

$$R_U = \begin{bmatrix} 0.192 & 0 & 0 & 0 \\ 0 & 0.09 & 0 & 0 \\ 0 & 0 & 0.16 & 0 \\ 0 & 0 & 0 & 4.25 \end{bmatrix} \qquad R_{\Delta U} = \begin{bmatrix} 0.1 & 0 & 0 & 0 \\ 0 & 0.077 & 0 & 0 \\ 0 & 0 & 0.2 & 0 \\ 0 & 0 & 0 & 0.83 \end{bmatrix}$$

Controller Tuning

All four controllers require extensive parameter tuning to ensure proper performance. The PID controller has eighteen gains, the LQR controller has sixteen costs, the LQI controller has nineteen costs, and the MPC controller has twenty weights associated with it. For this project, three tuning methods were evaluated: manual trial and error, the Ziegler–Nichols Method, and metaheuristic approaches.

Manual tuning by trial and error is a process that involves adjusting gains, costs, or weights iteratively, either one at a time or in groups, to finely tune the controller to increase performance. For simple controllers, such as ones used with uncoupled, low degree-of-freedom systems, manual tuning can be quite effective and can produce a controller that performs very well. However, for controllers associated with highly coupled systems or systems with many degrees-of-freedom, manual tuning can be time intensive and highly ineffective, if not impossible. It can be hard to tune parameter-by-parameter when one parameter may affect multiple system states or control inputs. For this project, manual trial and error tuning was

attempted but found ineffective as it was unable to tune any controller to the point of optimal performance.

The Ziegler–Nichols Method is a method that was developed to take the trial-and-error aspect out of manual tuning. It was designed for PID controllers but was investigated with the hopes of adapting it for the LQR, LQI, and MPC controllers. The specific Ziegler–Nichols Method researched was the frequency response method. This method works by setting the integral and derivative gains to zero and slowly increasing the proportional gains until associated states begin oscillating. This proportional gain is then defined as the “ultimate gain”, K_U, and the system’s period of oscillation is then defined as T_U. The gains for the PID controller are then determined as follows:

$$T_i = 0.5 * T_U \qquad T_d = 0.125 * T_U$$

$$K_p = 0.6 * K_U \qquad K_i = \frac{K_p}{T_i} \qquad K_d = K_p * T_d$$

Unfortunately, although effective for uncoupled or SISO systems, the Ziegler–Nichols Method is not overly effective for coupled, MIMO systems, such as a quadcopter. The Ziegler–Nichols Method assumes linearity between an input and an output and is not an effective solution for tuning complex, coupled systems. [20]

The third type of tuning method I studied for this project is metaheuristic methods. A metaheuristic refers to a high-level heuristic, or problem-solving strategy designed to optimize a result, which is made generic and used for a wide range of problems. Two popular metaheuristic algorithms used for tuning parameters are the particle swarm and genetic algorithm optimization methods. Both methods are referred to as population-based metaheuristic algorithms, meaning they perform optimization problem-solving techniques using a population of viable solutions, manipulating them over many iterations to ideally converge on a global minimum defined by a problem-specific fitness function.

The particle swarm optimization algorithm was inspired by group animal behaviors, such as bird flocks and schools of fish. The particle components of the algorithm refer to a potential solution, or a set of optimized gains in the case of this project. A particle has an associated position, or fitness function value, and a velocity, or its gradient in search space. The algorithm uses a user-defined fitness function, which was selected to be a six degree-of-freedom RMSE comprised of x, y, z, pitch, roll, and yaw error with respect to the reference trajectory. An optimized set of gains indicates a minimized RMSE. Each particle is affected at each iteration by its history and the global best position, or particle with the overall lowest RMSE. Each particle can see the global best position and the positions and velocity of its neighbors. With moderate random variations introduced along the way, a particle will update its parameters via a particle update equation as it moves closer to the best global position. Over many iterations, particles will begin converging towards a minimum RMSE as they mix and slide towards local iteration minimums defined by the global best position. The particle update equations are as follows:

X : Coordinate of particle

V : Velocity of particle

w : Inertia weight - modify impact of weight of previous velocity

P : Coordinate of particle's personal best

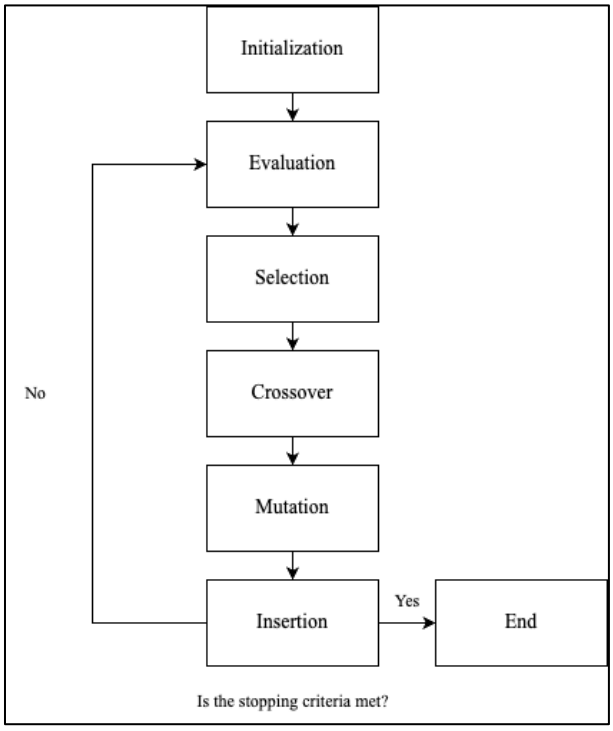
G : Coordinate of best global position

c_1 & c_2 : Weights on particle's personal best and best global position

r_1 & r_2 : Random values to introduce variation

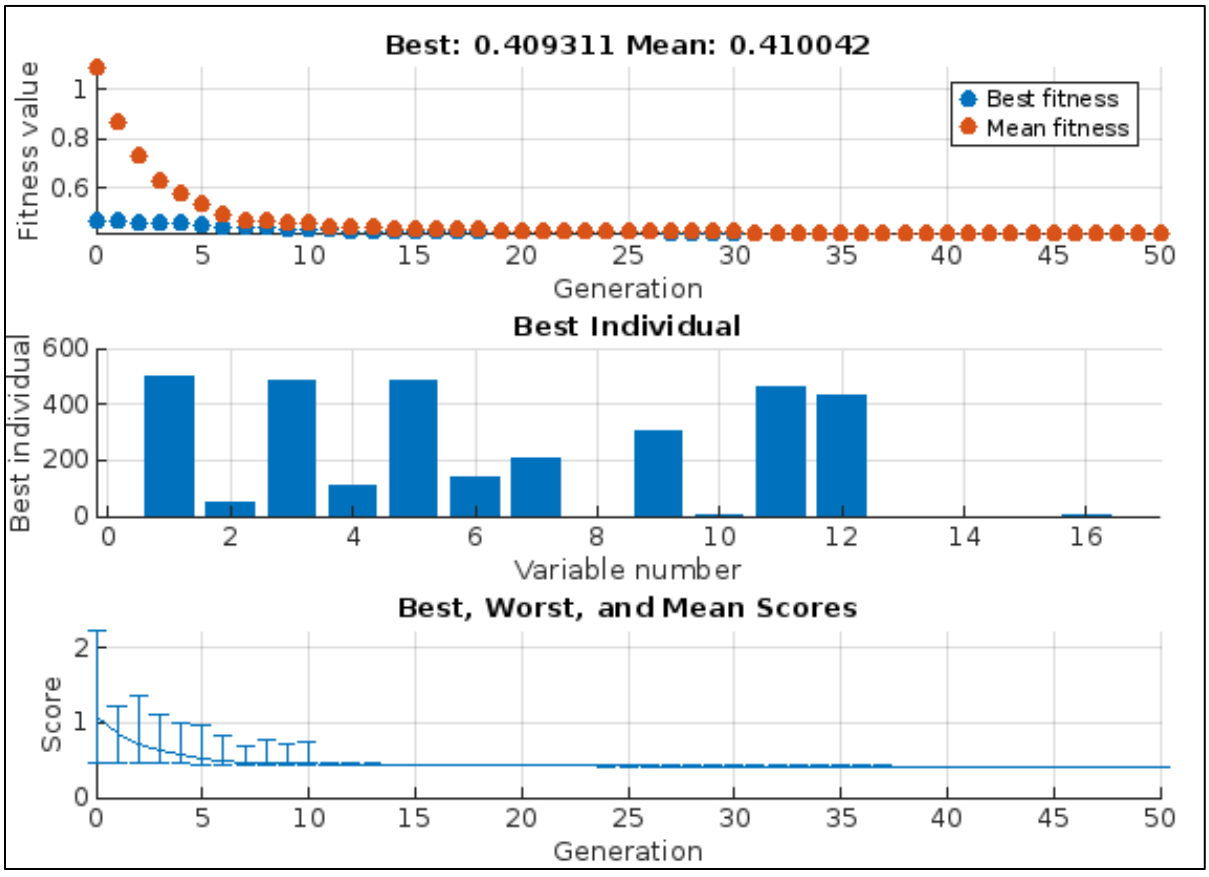
$$V^{i+1} = wV^i + c_1 r_1 (P - X) + c_2 r_2 (G - X)$$
$$X^{i+1} = X^i + V^{i+1}$$

The genetic algorithm is an optimization algorithm that was inspired by natural selection. The genetic algorithm begins with a population of potential solution candidates. The fitness function (RMSE) of each candidate in the population is evaluated at each iteration. Candidates with better fitness at each iteration are more likely to be selected to be part of the population for the next iteration than candidates with poor fitness. The population is re-populated through crossover, a method which involves the algorithm replacing the removed candidates with combinations and mixes of the successful candidates. Additionally, random variations, known as mutations, are introduced to the population to increase population diversity and search range. Candidate selection, crossover, and mutation steps are probabilistic processes. The new candidates are inserted into the population before further analysis. The algorithm iterates and slowly evolves towards candidates with better fitness values until termination criterion is met. Termination criterion could be several parameters, such as maximum number of interactions, convergence tolerance, or stalling. The general process of the genetic algorithm is as follows:



Both optimization algorithms perform very well. Particle swarm optimization typically converges to a minimum value faster than the genetic algorithm, as the genetic algorithm is computationally extensive algorithm. Additionally, the particle swarm algorithm can be easier to implement with less parameters and components. However, the genetic algorithm has a better solution exploration capability and can manage problems with large populations and parameters. The genetic algorithm is also more likely to converge to a global minimum, as opposed to a local minimum, than the particle swarm algorithm due to its robustness, random candidate mutations, and a constantly changing population set. [22] [23]

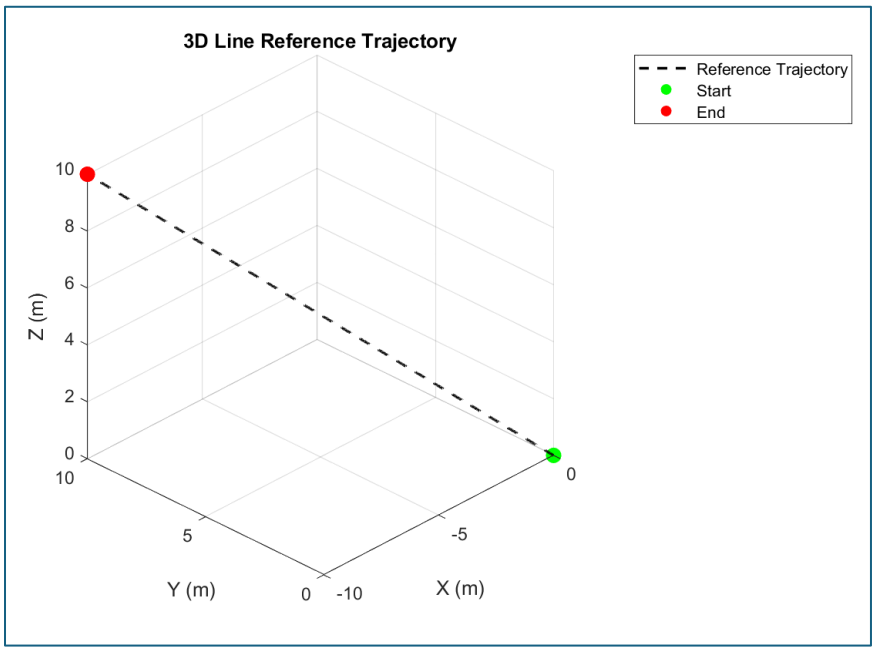
For this project, MATLAB's genetic algorithm function, *ga*, in the Global Optimization Toolbox was leveraged. The controllers were packaged as RMSE cost functions and passed through the genetic algorithm function to optimize the gains, costs, or weights of the controller. After termination, the *ga* function outputs the optimized controller parameters to the MATLAB terminal. For this project, the genetic algorithm optimization problem for each controller began with a randomized initial population to ensure search capabilities were not limited. The optimal output gains were reintroduced to the genetic algorithm as a seed candidate in the population so the algorithm would further optimize the performance. A visualized example of fitness function convergence and genetic algorithm outputs available with the MATLAB *ga* function when applied to a LQR controller can be seen as follows [24]:

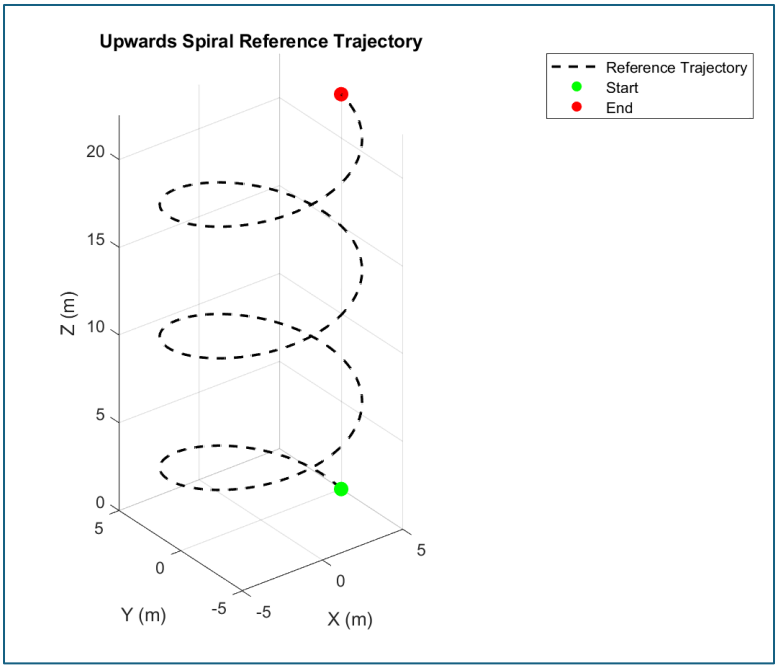


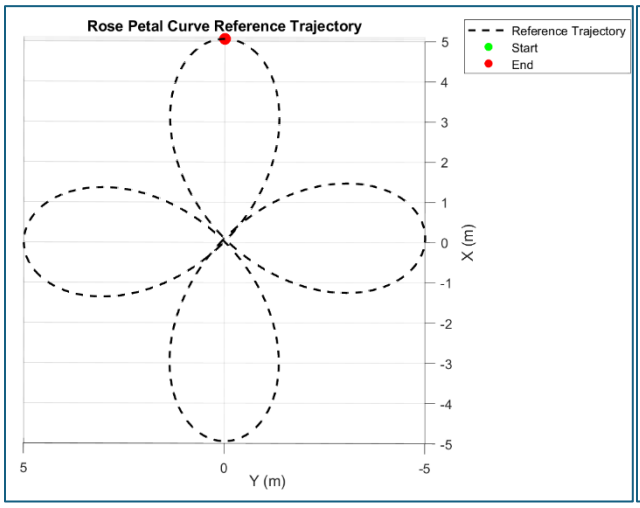
Simulations

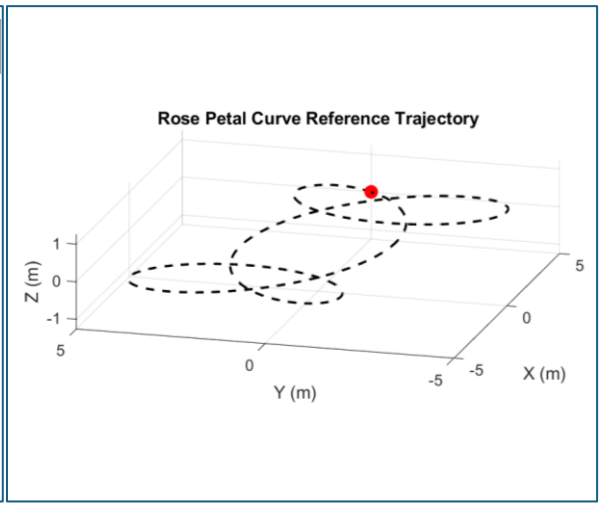
Simulation Setup

The controller and system were simulated against varying reference trajectories in MATLAB. Three trajectories were defined: a 3D line, an upwards spiral, and a rose petal curve. The 3D line was meant to evaluate the controller's ability to keep the system on a simple reference trajectory that involved a straight path. The trajectory was designed as a ten second straight line. The upwards spiral was meant to evaluate the controller's ability to keep the system on a slightly more complicated trajectory, introducing pitch and roll inputs to move the UAV in a circular pattern while climbing. The trajectory was designed as a thirty second climbing spiral. The rose petal curve was meant to evaluate the controller's ability to keep the system on a complex trajectory involving multiple direction and altitude changes. The trajectory was designed as one full rose petal curve over the course of 30 seconds.









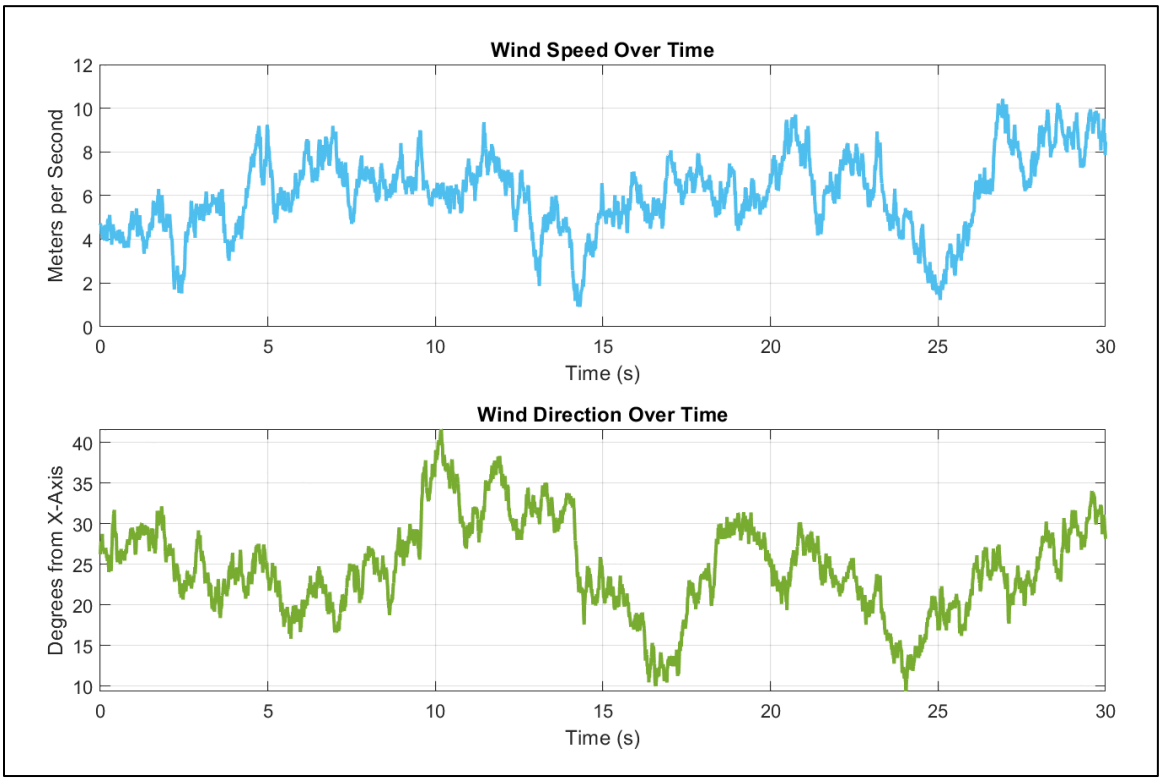
The controllers were defined on the 3D line trajectory, then further tuned them on the spiral before evaluating them on both trajectories. This was done to ensure the controller was capable of handling simple translations as well as sinusoidal paths. The controllers were then evaluated on the rose petal trajectory to determine their robustness. An unmeasured disturbance was then introduced to the system, simulating a varying wind gust pushing in the x-y plane. Modeling wind gusts was an appropriate and realistic external unmeasured disturbance to simulate for a quadcopter, and this was intended to evaluate the system's ability to reject constant external disturbances not accounted for in the system model.

The wind was modelled using a Gaussian distribution with autoregressive smoothing. The mean wind value was five meters per second, with a standard deviation of two meters per second. The mean wind direction with respect to the x-axis was 25 degrees, with a standard deviation of 5 degrees. Using the Gaussian distribution function *randn()* in MATLAB, and an

autoregressive smoothing equation, a smoothed wind speed and angle noise value was generated at each iteration [22]:

$$y = \text{Noise value}$$
$$\sigma_y = \text{Noise standard deviation}$$
$$\alpha = \text{Smoothing factor } (0.99)$$
$$r = \text{Random value from Gaussian distribution}$$

$$y^i = \alpha y^{i-1} + \sigma_y r \sqrt{1 - \alpha^2}$$



Using the wind direction and speed, the wind values were split into x- and y-magnitude components with respect to time. Quadcopter aerodynamic parameters were estimated from general knowledge of UAVs and the x- and y-components of the force, *F,* applied to and resultant acceleration, *a*, of the quadcopter from the wind were calculated:

$$m = 1.25\ \text{kg} \quad C_d = 0.75 \quad A = 0.1m^2 \quad \rho = 1.225\frac{\text{kg}}{m^3}$$

$$F = \frac{1}{2}\rho A C_d v^2 \quad a = \frac{F}{m}$$

Performance Metrics

To quantitatively determine each controller's performance during its flight, computation time, one-dimensional RMSE values (individual x-, y-, and z-errors), three-dimensional RMSE values (combined x, y, and z error), and flight smoothness quantified as oscillatory behavior analysis were chosen as comparative performance metrics.

The computation time metric tracked the amount of time the simulation took to run. Although this metric does not indicate real-world performance and may vary depending on the computer used, it does shed light on the relative computational expense of each controller.

The RMSE values for the x, y, and z states, as well as the 3-dimensional RMSE, over time were calculated as follows:

$$e_x(i) = X[1](i) - X_{ref}[1](i), \quad e_y(i) = X[3](i) - X_{ref}[3](i), \quad e_z(i) = X[5](i) - X_{ref}[5](i)$$

$$RMSE_x = \sqrt{\frac{1}{N}\sum_{i=1}^{N} e_x(i)^2}, \quad RMSE_y = \sqrt{\frac{1}{N}\sum_{i=1}^{N} e_y(i)^2}, \quad RMSE_z = \sqrt{\frac{1}{N}\sum_{i=1}^{N} e_z(i)^2}$$

$$RMSE_{3D} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} [e_x(i)^2 + e_y(i)^2 + e_z(i)^2]}$$

This scalar performance metric enables comparison of both the overall positional error value of the quadcopter with respect to its reference trajectory, but also positional errors on individual axes. Unfortunately, if a particular controller type results in a path that oscillates around a reference trajectory, the RMSE for that controller may be similar to a controller that follows the reference trajectory more smoothly but with a small constant offset. This was observed when comparing the PID controller paths, which oscillated slightly around the reference trajectory, with the smoother LQR, LQI, or MPC controllers. To differentiate between controllers that produce paths with similar RMSE values but vastly distinct levels of smoothness, numerical oscillatory behavior characterization was used.

To numerically verify the observed oscillations, the curvature of each 3-dimensional trajectory was calculated. The equation for the curvature of a 3-dimensional trajectory is as follows [25]:

$$X(t) = \text{Trajectory}, \quad \kappa = \text{Curvature}$$

$$\kappa = \frac{\left\| \dot{X}(t) \times \ddot{X}(t) \right\|}{\left\| \dot{X}(t) \right\|^3}$$

MATLAB's *findpeaks* function was leveraged, using the calculated curvature of the 3-dimensional trajectory as the evaluated function, to determine the number of local peaks given a user-defined threshold. The number of peaks in the function provides numerical insight into the

observed oscillatory behavior, with a lower number of peaks indicating a smoother trajectory. [26] [27]

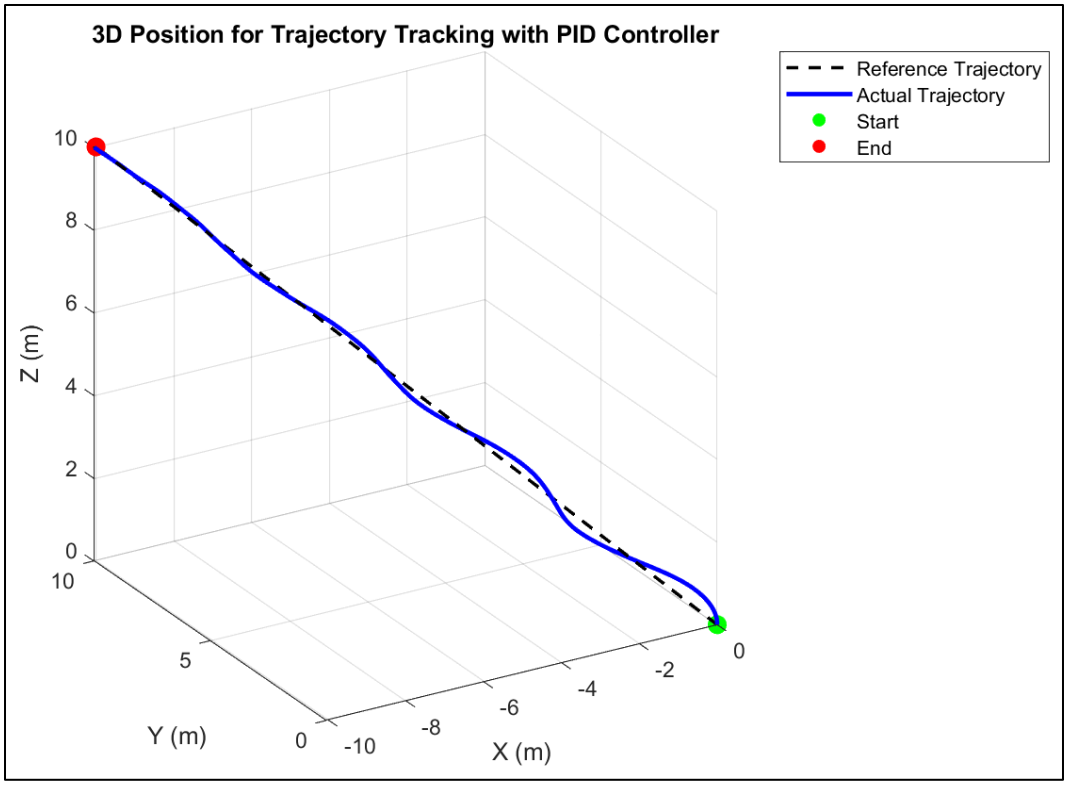
Results

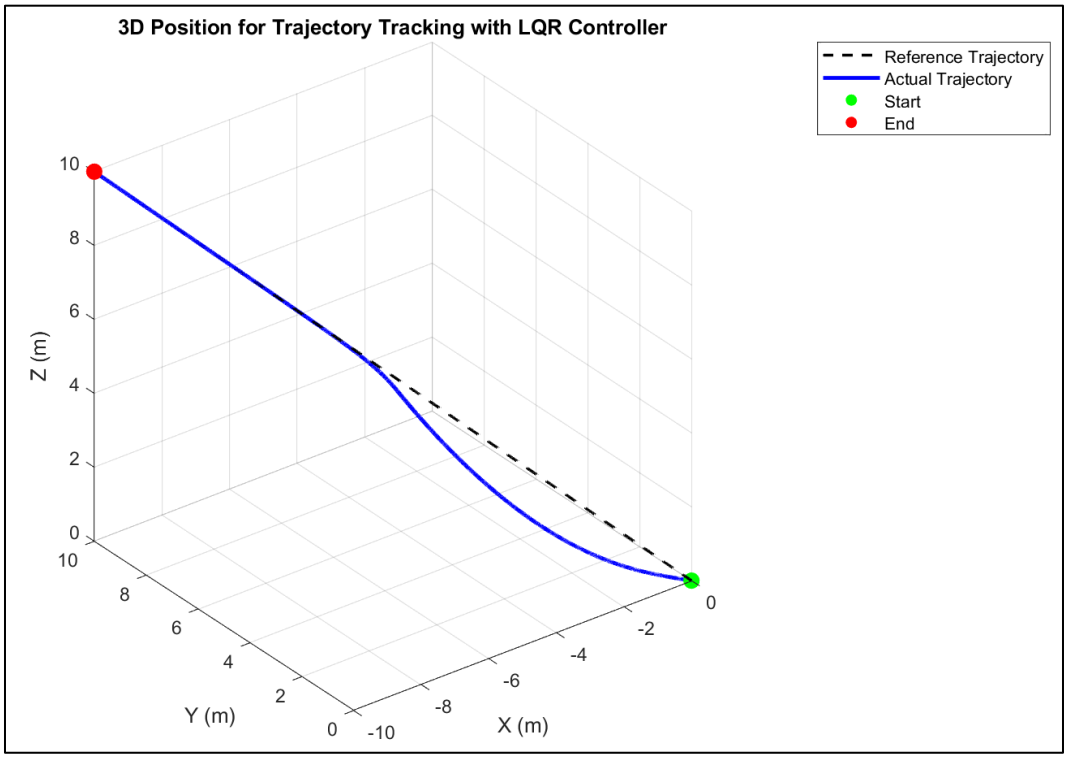
All four controllers were evaluated with the same quadcopter model on the three unique trajectories to assess the controller's performance and versatility in controlled conditions. This was followed by retesting each of the four controllers on the rose petal curve with the simulated wind added as a varying, unmeasured disturbance to test the controller's ability to reject external disturbances while it tracked a complex trajectory.

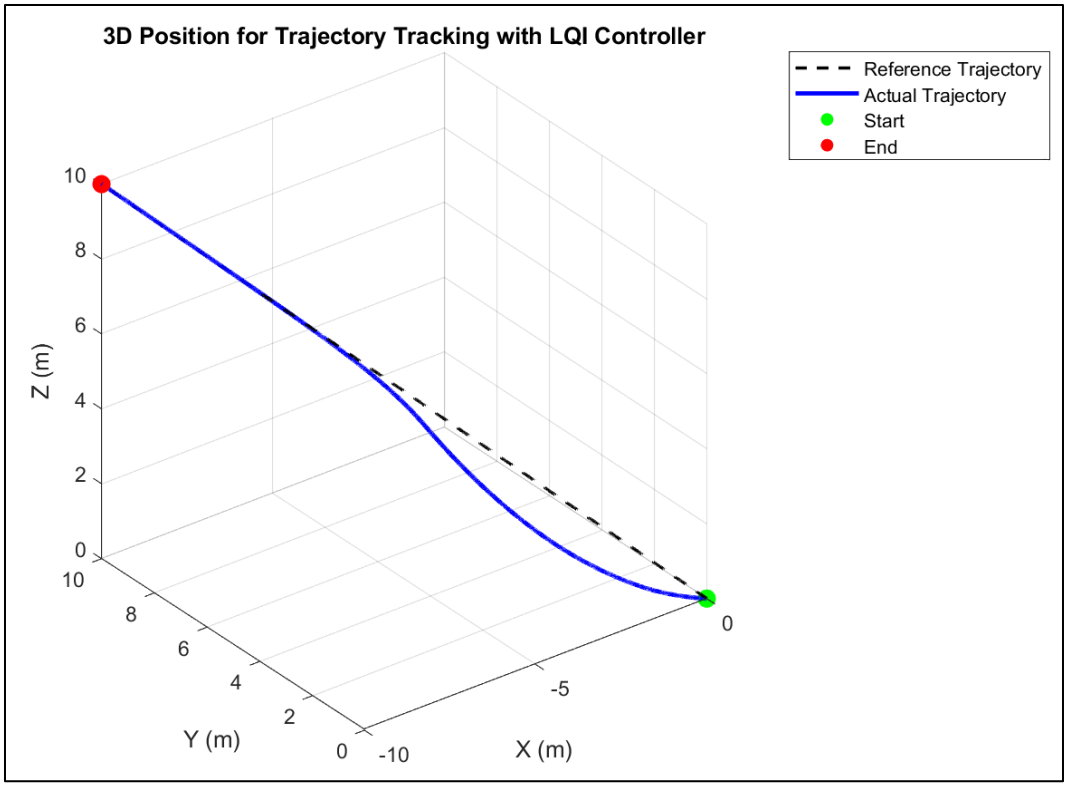
Without wind being introduced to the environment, the four controllers performed as follows on the 3D line trajectory with the best and worst values indicated with green and red colors, respectively:

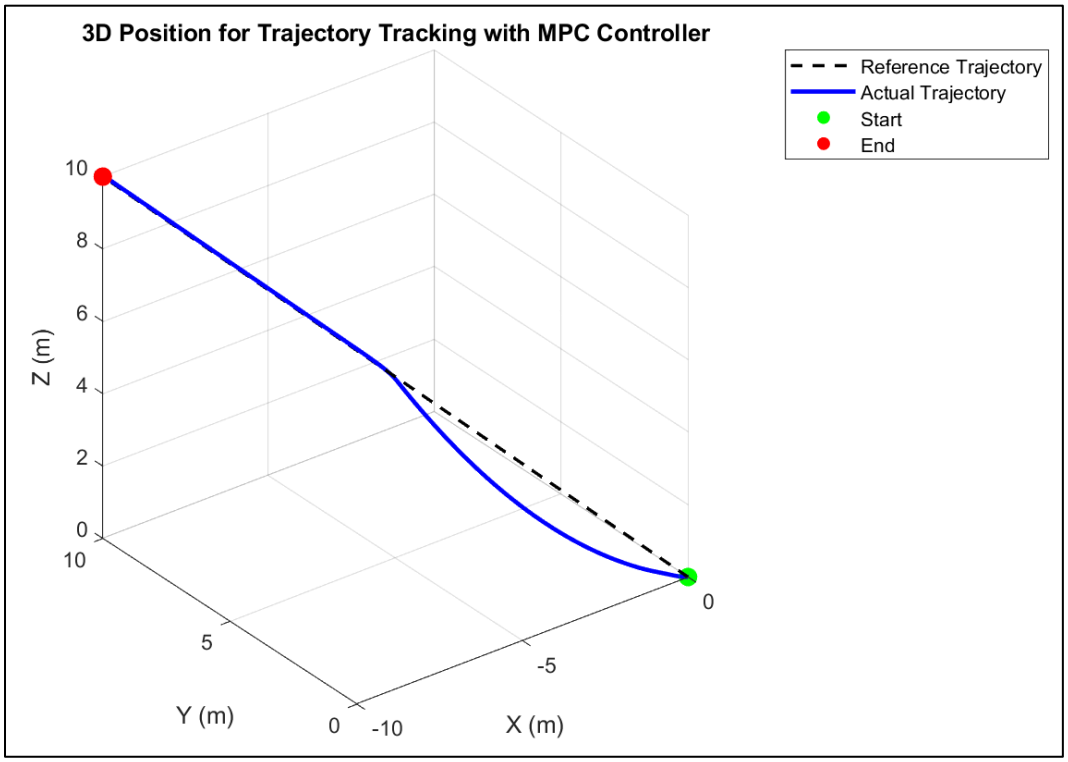
3D-Line Trajectory				
	PID	LQR	LQI	MPC
Time (s)	0.013	0.012	0.023	0.470
RMSE – 3D (m)	0.360	0.702	0.704	0.716
RMSE – X (m)	0.330	0.041	0.022	0.106
RMSE – Y (m)	0.143	0.034	0.033	0.096
RMSE – Z (m)	0.014	0.700	0.702	0.702

In this simulation, all four controllers completed the simulation quickly, which was expected given the simple trajectory and short duration, but the MPC performed significantly slower with respect to the other three times. The LQR, LQI, and MPC controllers performed similarly with respect to RMSE and had great accuracy in the x- and y-directions but lagged at the beginning in the z-direction due to damped control inputs. The PID controller was significantly more aggressive and closed the gap in the z-direction quickly but oscillated more, which led to increased x- and y-direction errors. The following plots show the controller's performance in 3D space, and additional plots showing control inputs and individual degree-of-freedom trajectories can be found in *Appendix A*:


3D Position for Trajectory Tracking with PID Controller
Reference Trajectory
Actual Trajectory
Start
End
Z (m)
Y (m)
X (m)


3D Position for Trajectory Tracking with LQR Controller
Reference Trajectory
Actual Trajectory
Start
End
Z (m)
Y (m)
X (m)

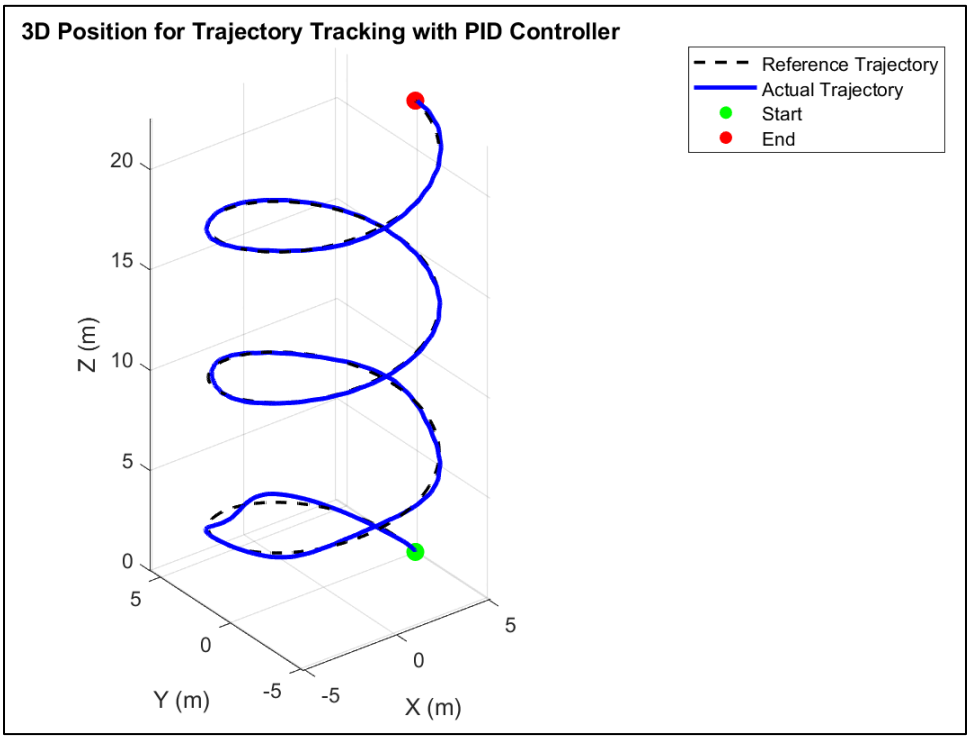


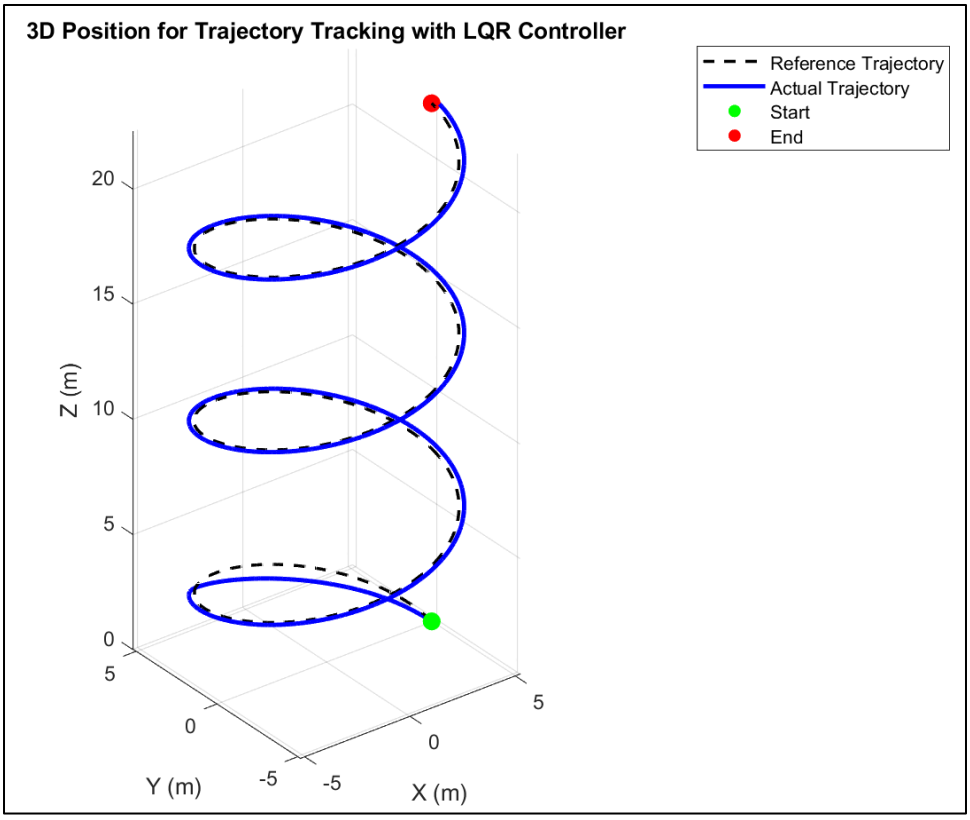


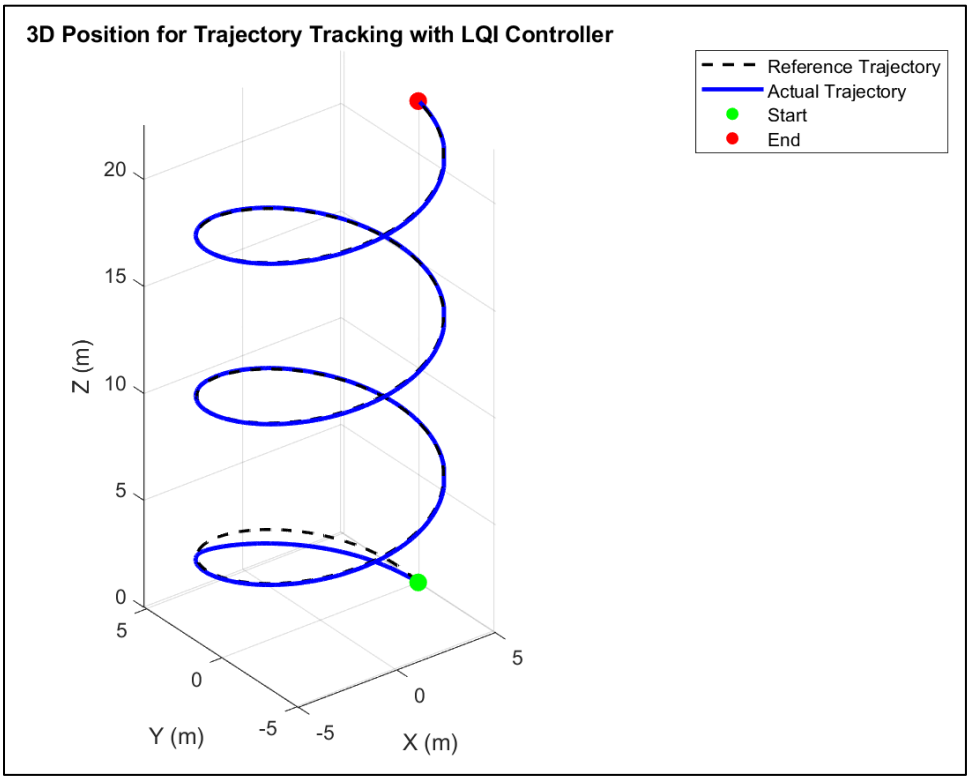
The performance metrics for all four controllers without wind added on the upwards spiral trajectory, a longer more complex path compared to the line, with the best and worst values indicated with green and red colors, respectively, are as follows:

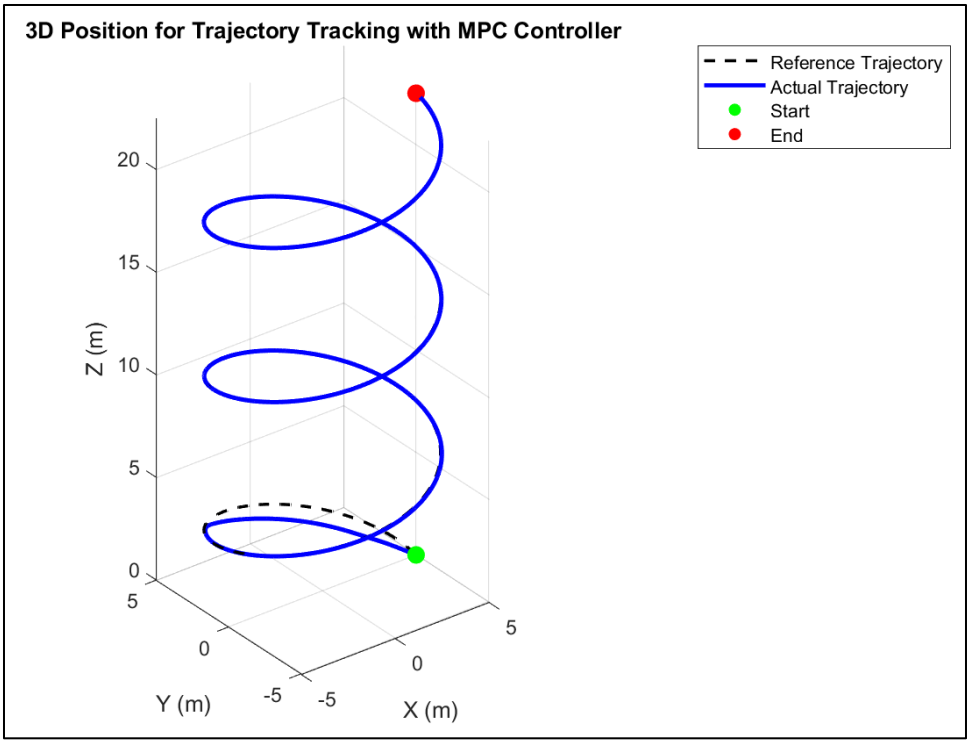
Upwards Spiral Trajectory				
	PID	LQR	LQI	MPC
Time (s)	0.017	0.015	0.021	1.183
RMSE – 3D (m)	0.301	0.307	0.234	0.343
RMSE – X (m)	0.106	0.177	0.044	0.181
RMSE – Y (m)	0.282	0.156	0.115	0.209
RMSE – Z (m)	0.006	0.198	0.199	0.203

This simulation was more complex and three times as long as the 3D line trajectory, and although the MPC controller's computation time increased dramatically, the PID, LQR, and LQI controllers were able to complete the simulation incredibly fast due to their computational efficiency. The PID controller, although fast, struggled with oscillations around the trajectory and did not settle into a smooth spiral. The LQR and MPC controllers struggled with a constant offset throughout the path as they drifted around the spiral too widely, leading to increased positional RMSE throughout the flight. The LQI controller was very accurate and was able to better reject the constant offset the LQR and MPC controllers experiences due to its integral action components. The following plots show each controller's performance in 3D space, and additional plots showing control inputs and individual degree-of-freedom trajectories can be found in *Appendix A*:




3D Position for Trajectory Tracking with LQR Controller
Reference Trajectory
Actual Trajectory
Start
End
Z (m)
20
15
10
5
0
5
0
-5
Y (m)
-5
0
5
X (m)


3D Position for Trajectory Tracking with LQI Controller
Reference Trajectory
Actual Trajectory
Start
End
Z (m)
20
15
10
5
0
5
0
-5
Y (m)
-5
0
5
X (m)



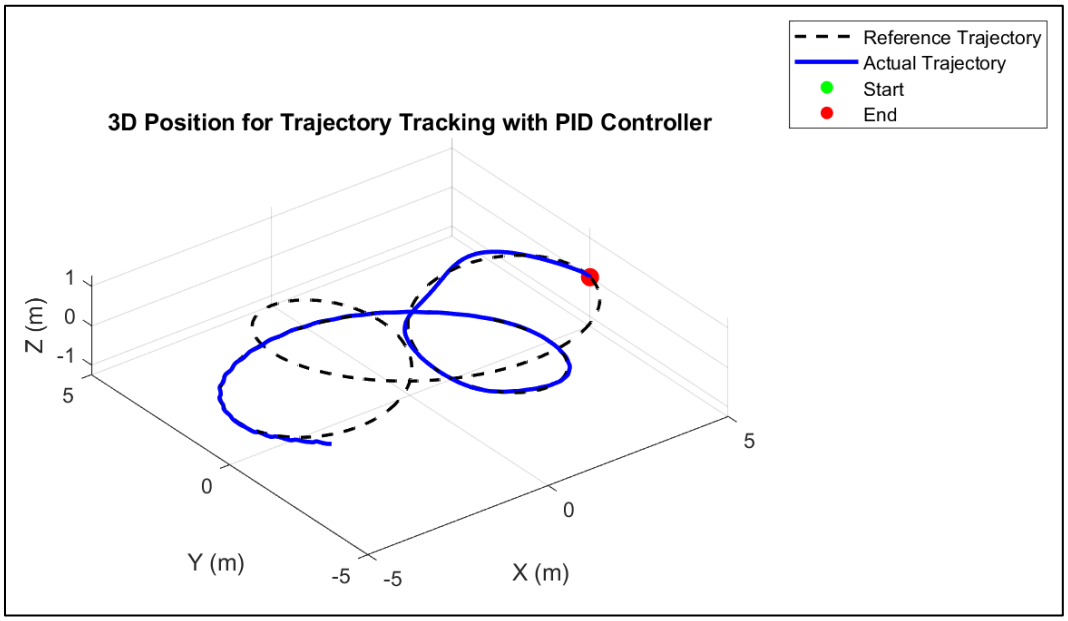
Unfortunately, the growing oscillations seen with the PID controller on the line and upwards spiral trajectories were too much for the model to manage on the rose petal curve trajectory. The PID controller was able to take the quadcopter model about halfway before the model diverged and the oscillations became too much. The quadcopter ended up flipping over and was unable to complete the trajectory. The PID controller paired with the nonlinear quadcopter model used was a complicated and sensitive system, and although extensive tuning optimization enabled the controller to complete the line and spiral trajectories, the system was too aggressive and sensitive to complete the more complex rose petal curve.

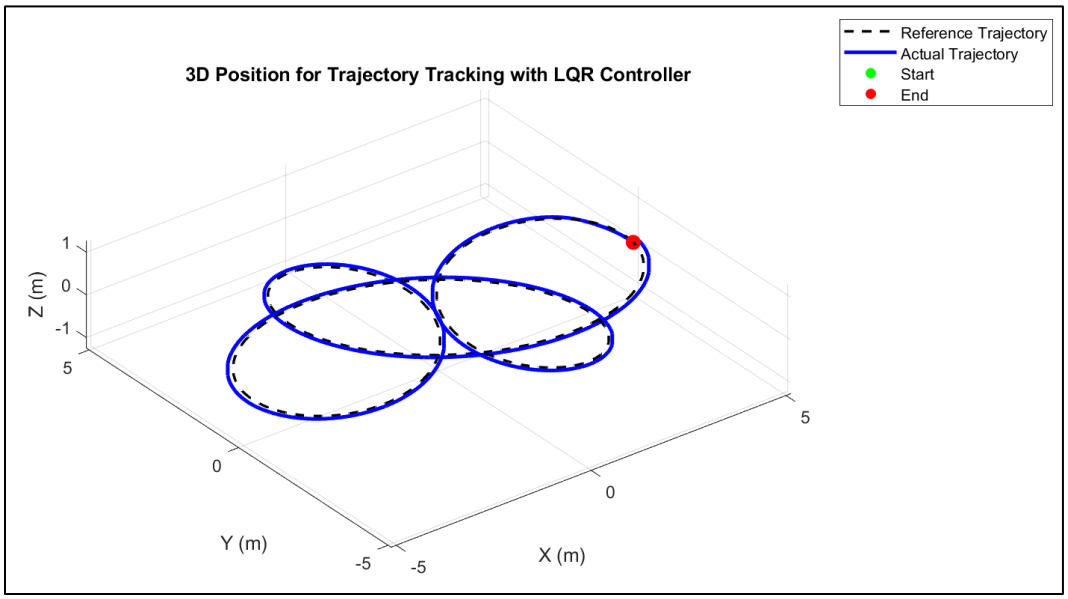
The performance metrics for the remaining three controllers without wind added on the rose petal curve trajectory, a trajectory with the same duration but increased complexity compared to the upwards spiral, with the best and worst values indicated with green and red colors, respectively, are as follows:

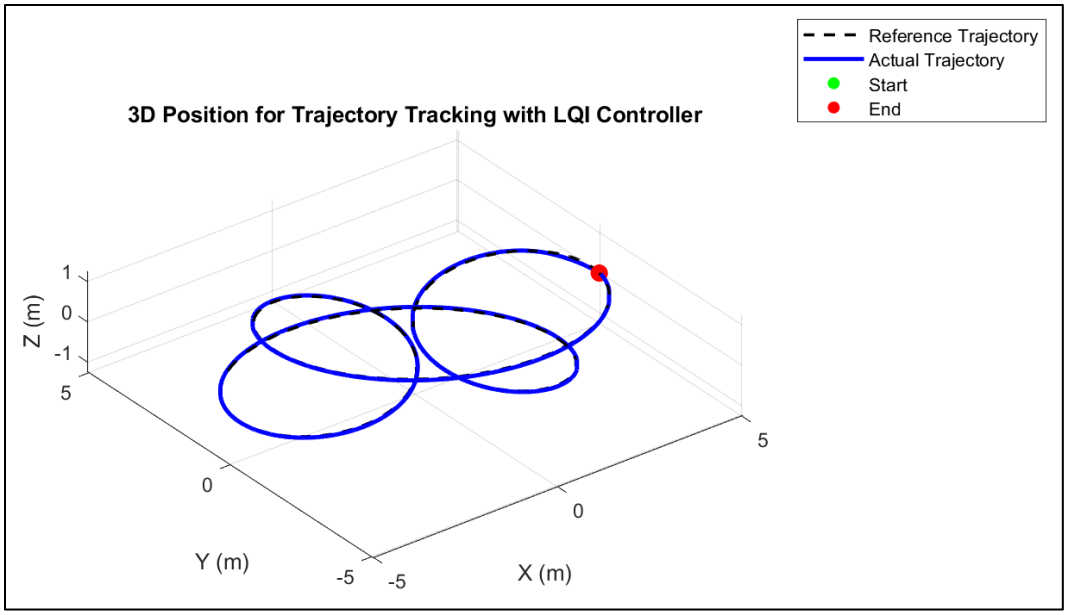
Rose Petal Curve Trajectory				
	PID	LQR	LQI	MPC
Time (s)	N/A	0.021	0.022	1.193
RMSE – 3D (m)	N/A	0.114	0.056	0.147
RMSE – X (m)	N/A	0.088	0.022	0.101
RMSE – Y (m)	N/A	0.071	0.050	0.105

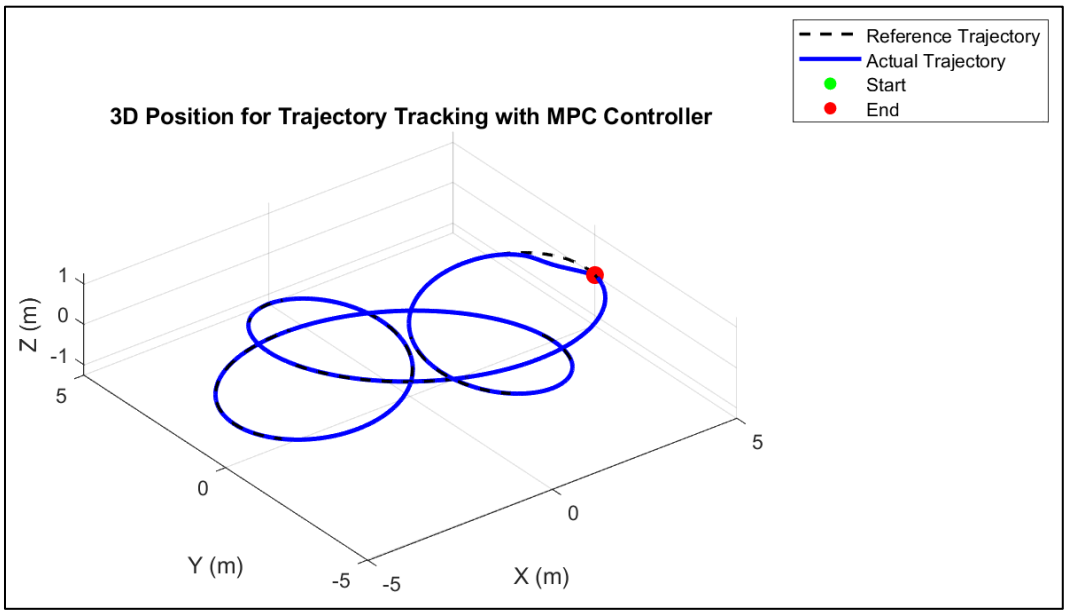
RMSE – Z (m)	N/A	0.014	0.015	0.019

The MPC continued to be a slower controller compared to the LQR and LQI controllers due to its computational intensity. Unfortunately, even with the increased computation time, the MPC controller performed worse than the LQR and LQI controllers in all RMSE categories. Its error was closer to the LQR controller's, though, and both struggled with managing the persistent offsets caused by the frequent direction changes and complex curvature of the rose petal curve trajectory. The LQI controller performed very well, being both quick and accurate. The following plots show the controller's performance in 3D space, and additional plots showing control inputs and individual degree-of-freedom trajectories can be found in *Appendix A*:









The path smoothness metric quantified as oscillatory analysis, or the number of peaks that arise in the curvature equation, for all four controllers on the three paths with no external disturbances is as follows:

Oscillatory Behavior Analysis – Number of Peaks				
	PID	LQR	LQI	MPC
3D-Line	20	1	2	2
Upwards Spiral	43	1	2	1

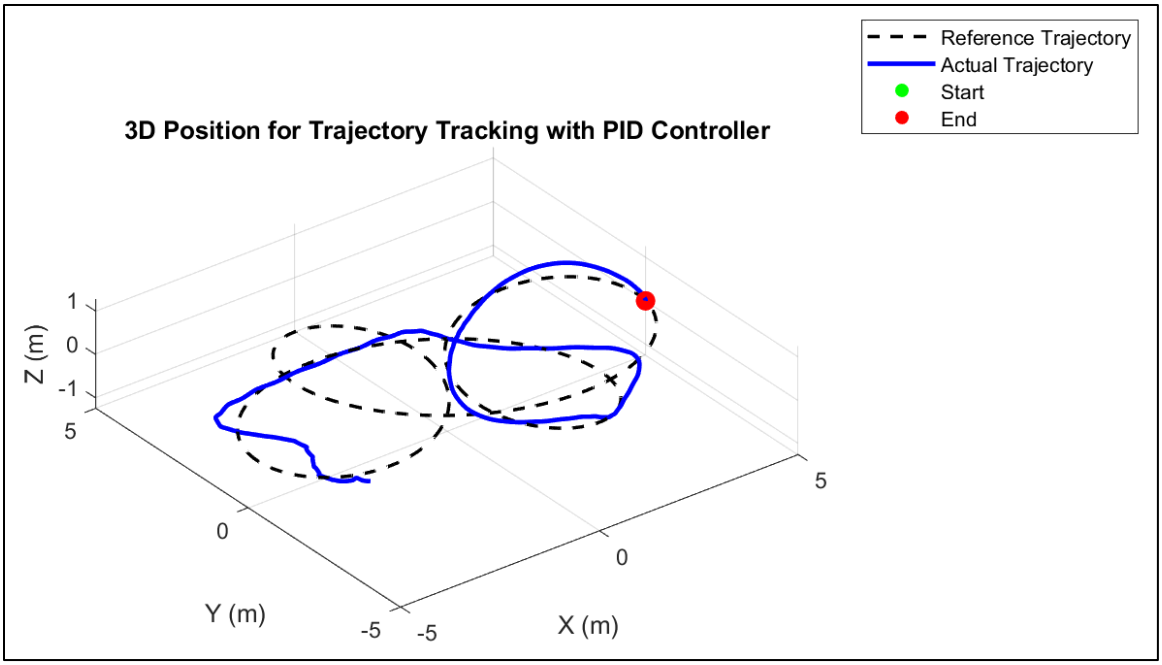
Rose Petal Curve	48	6	5	3

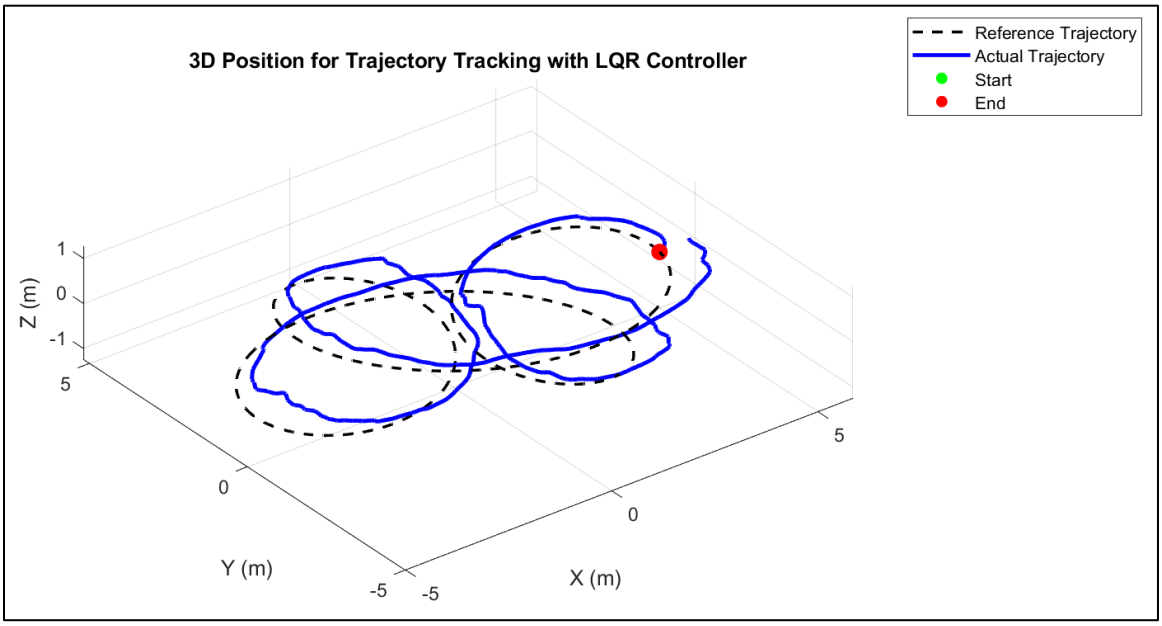
Even when the PID controller seemed to perform well with respect to RMSE accuracy and computation time, the resultant trajectory of the quadcopter is not ideal. Many peaks in the curvature equation indicate significant oscillatory behavior and a lack of trajectory smoothness. The PID controller's resultant trajectories were incredibly rough and would not be ideal for flight operations despite often having similar RMSE values to the other controllers. The LQR, LQI, and MPC controllers all performed similarly with respect to smoothness, indicating smooth and controlled trajectories during the duration of the flight.

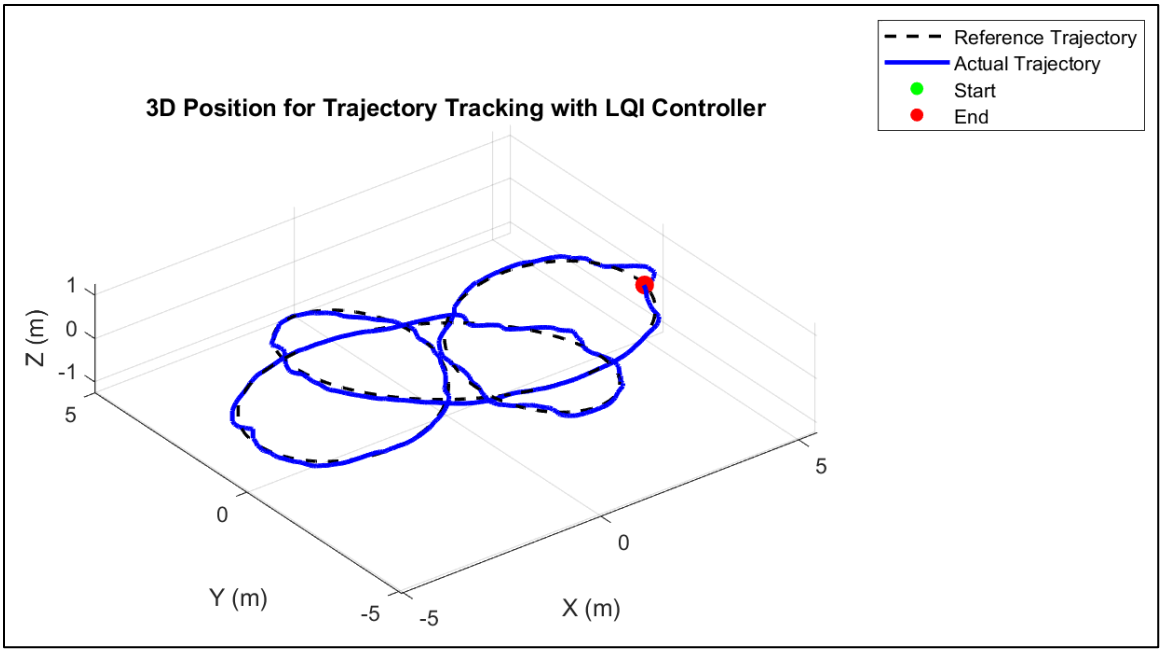
Given that the PID controller was unable to track the rose petal curve without added disturbances, its inability to track the rose petal curve with added external disturbances modeled as a varying cross wind was expected. The RMSE performance metric was used to compare the performance of the controllers when faced with external unmeasured disturbances. Because the simulated wind is not constant and uses a Gaussian distribution to simulate its speed and angle, the disturbance is not constant between simulations. To account for this, the values of the five simulations were averaged. The performance metrics for the remaining three controllers with wind added as an external, unmeasured disturbance on the rose petal curve trajectory with the best and worst values indicated with green and red colors, respectively, are as follows:

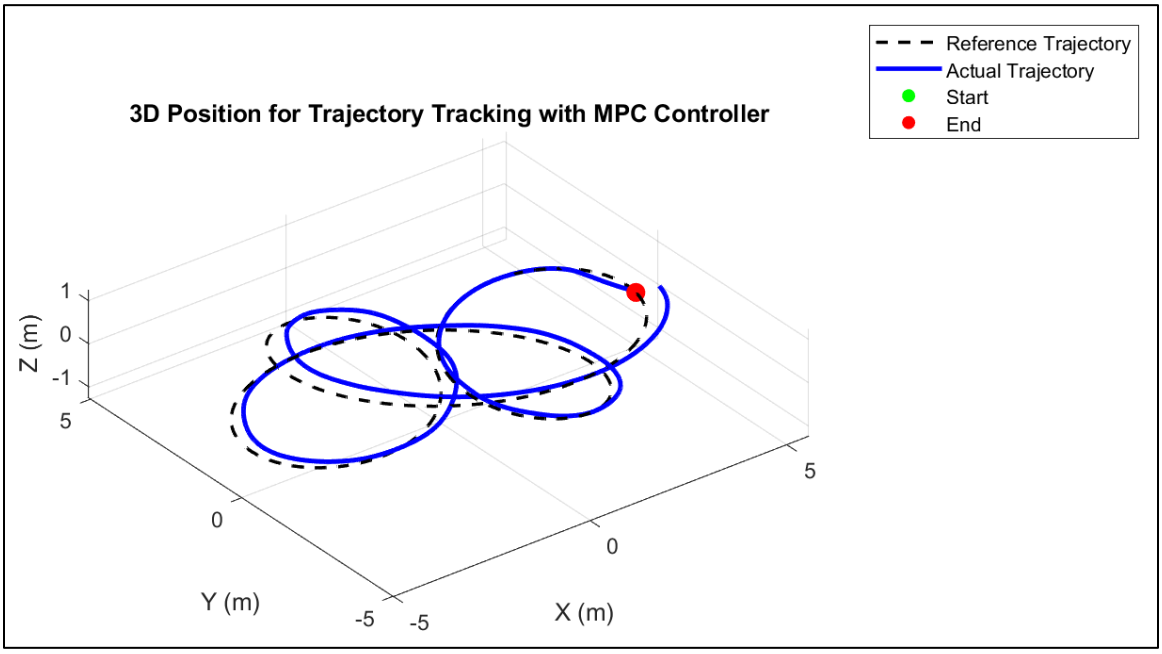
Rose Petal Curve Trajectory with Wind				
	PID	LQR	LQI	MPC
RMSE – 3D (m)	N/A	0.816	0.167	0.330
RMSE – X (m)	N/A	0.797	0.149	0.311
RMSE – Y (m)	N/A	0.170	0.073	0.110
RMSE – Z (m)	N/A	0.17	0.015	0.019

Disregarding the PID controller's inability to track the rose petal curve trajectory, this simulation highlighted the capability of each controller to reject the constant external disturbance. The LQI controller performed very well, responding to the noise well and tracking the trajectory accurately despite some roughness. The LQR controller had a similar resultant path, but with a constant offset caused by the wind that the controller was unable to reject. The MPC controller had similar difficulties with the constant disturbance pushing the quadcopter off track but managed it better than the LQR controller did. The following plots show the controller's performance in 3D space, and additional plots showing control inputs and individual degree-of-freedom trajectories can be found in *Appendix A*:


Reference Trajectory
Actual Trajectory
Start
End
3D Position for Trajectory Tracking with PID Controller
Z (m)
1
0
-1
5
0
Y (m)
-5
-5
0
5
X (m)


Reference Trajectory
Actual Trajectory
Start
End
3D Position for Trajectory Tracking with LQR Controller
Z (m)
1
0
-1
5
0
Y (m)
-5
-5
0
5
X (m)





Discussions

Overall, the LQI controller performed the best when comparing computation time, accuracy, and smoothness together, even though the LQR and MPC controllers also performed very well. It required little computation time and had excellent positional accuracy and smooth resultant paths with all trajectories. It also managed external unmeasured disturbances better than the other three controller types. The LQR controller performed similarly to the LQI

controller in most metrics but expectedly was unable to manage constant errors or external disturbances as the LQI controller did. The MPC controller performed similarly to the LQR controller, but with significantly greater computation times. The PID controller was aggressive with little delays in matching the reference trajectory but was unstable.

The PID controller, despite its inability to track complex trajectories well, had a more aggressive response to state errors than the other three controllers. Specifically in the z-direction, it rose to meet the trajectory very quickly. Unfortunately, it also quickly developed angular oscillations that led to the quadcopter straying off path and flipping over when tested on the rose petal curve. In real operating environments, a quadcopter may have a weapon or sensor on it that is sensitive to excessive motion, a lightweight structure that is susceptible to breaking under rapid oscillations, or power restrictions that would be exceeded with the required battery pull to handle and stabilize rapid oscillations. A quadcopter that oscillates rapidly, despite its aggressive trajectory tracking behaviors and acceptable RMSE values, is unsuitable for real-life conditions and operations. The controller performed very well when it was specifically tuned to a trajectory, but when the gains were optimized along varying trajectories in hopes of generalizability, the controller struggled. If this controller were to be implemented on a real quadcopter, it would likely require multiple sets of gains for different trajectory types to ensure stability.

Despite relying on more complex mathematical theory as a backing for the controllers when compared to the PID controller, the LQR, LQI, and MPC controllers were easy to implement in MATLAB, whereas the PID controller required extensive integration and frame conversions with a nonlinear system model. The PID controller, though, as well as the LQR and LQI controllers, were less computationally expensive compared to the MPC controller during simulation and ran significantly faster.

When wind was introduced to the simulation as a constant external unmeasured disturbance, the LQI controller performed very well, minimizing RMSE over the course of the flight path compared to the other three controllers. The PID controller diverged and was unable to complete the trajectory due to oscillations. The LQR and MPC controllers successfully tracked the rose petal curve but struggled with constant offsets due to the wind. A constant offset was expected with the LQR and MPC controllers, when compared to the LQI controller, due to the lack of integral action limiting the controller's ability to aggressively correct for constant offsets like the LQI controller is designed to do.

With the wind added as a varying disturbance, the PID, LQR, and LQI controllers exhibited a rough flight path due to the varying wind speeds introduced at each step. The MPC controller, however, exhibited an exceptionally smooth path. The setup of the PID, LQR, and LQI controller code allowed me to add an acceleration at each time step associated with the wind values at that time, directly introducing unmeasured disturbances to the system. The controllers are not designed to predict future control inputs that can manage random noise inputs, as the MPC controller is. MATLAB's *mpc* library was leveraged to set up the MCP controller, which limited how wind was able to be introduced compared with the other controllers. With the *mpc* library, noise needed to be introduced as an unmeasured disturbance, modeled as an additional set of system states. The MPC controller, due to its damped behavior,

well-tuned gains, and predictive control input calculations, was able to manage noise at each time step very well. It completed the rose curve trajectory smoothly and fully but was unable to fully track the trajectory without an offset due to its lack of integral action. If integral action was added to the MPC controller, and computation time was not of concern, the controller likely would have been the best option.

Summary and Future Work

In this project, a full six degree-of-freedom quadcopter was mathematically modeled and PID, LQR, LQI and MPC trajectory-tracking controllers were designed and implemented. Assumptions that include a near-hover state during flight and a linearized system model were leveraged for system modeling. The controllers were tuned on a 3-dimensional line trajectory and an upwards spiral trajectory, then evaluated on those two reference trajectories as well as a more complex rose petal curve. Following that, a varying wind was introduced as a constant, unmeasured, external disturbance to the system to evaluate the controller's versatility and ability to reject constant disturbances along a reference trajectory. The results indicated that given the design of these four controllers and the quadcopter model, the LQI is the best controller for generalized quadcopter trajectory tracking with varying external disturbances, such as wind. However, the MPC controller showed promise for scenarios where computation time is not of concern, and a similar MPC controller with added integral action may end up being the best option in future work.

This project was a very thorough introduction to trajectory tracking with PID, LQR, LQI, and MPC controllers. The quadcopter system was successfully modeled, and three loitering trajectories were successfully designed for testing. The project involved a thorough introduction to optimized tuning methods and successfully leveraged a genetic algorithm to tune the controllers faster and better than by manual trial and error. Simulated wind effects were accurately modeled and integrated into the simulation environment as an external disturbance. Using computation time, RMSE, and path smoothness performance metrics, controller performance comparison was thorough, and the LQI controller's superior performance for a quadcopter simulation of this nature was determined.

In future work, there are a few things that would be interesting advancements to the work done in this project. First, modifying these controllers to be able to integrate with a real quadcopter to compare simulation results with experimental results would offer valuable insight into controller performance. Second, adding integral action to the MPC controller to make a fifth controller type and evaluating its ability to reject constant disturbances would be fascinating. The standard MPC controller performed similarly to the LQI controller in most cases, and the addition of integral action would likely make the MPC controller perform similarly to or better than the LQI controller. Finally, evaluating the linear LQR, LQI, and MPC controllers on the nonlinear model of the quadcopter would be a great test of a controller's ability to track a trajectory in a realistic environment is to see if it can manage the nonlinear dynamics of a real system in simulation.

GitHub Code Repository

The entire set of MATLAB source code created for this project can be found here: https://github.com/dhartnet/ENPM808_Quadcopter_Controllers

Acknowledgments

I would like to thank Dr. Waseem Malik from the University of Maryland, College Park, for their willingness to be my advisor for this project despite their incredibly busy schedule. Their guidance, feedback, advice, and thoughtful conversations helped steer me through this project and introduced me to numerous fascinating and complex control concepts. I had a fantastic two years in the Maryland Applied Graduate Engineering (MAGE) Robotics Engineering program, and this semester project was a wonderful way to cap it off!

References

[1] Bayisa, Ayele Terefe. “Controlling Quadcopter Altitude Using PID-Control System.” *International Journal of Engineering Research And*, vol. V8, no. 12, 16 Dec. 2019, www.ijert.org.

[2] Sahrir, Nur Hayati. Basri, Mohd Ariffanan Mohd. “Modelling and Manual Tuning PID Control of Quadcopter.” *Control, Instrumentation and Mechatronics: Theory and Practice*, LNEE, vol. 921, pp. 346-357, 07 Jul. 2022, https://link.springer.com/chapter/10.1007/978-981-19-3923-5_30.

[3] Safarov, T.A.. “MATLAB Simulation of Quadcopter Dynamics and PID Attitude Control.” *Technium*, vol. 18, pp. 82-91, 2023, www.techniumscience.com.

[4] Mellinger, Daniel Warren. “*Trajectory Generation and Control for Quadcopters.*” PhD in Mechanical Engineering and Applied Mechanics, University of Pennsylvania, 2012. Penn Libraries, http://repository.upenn.edu/edissertations/547?utm_source=repository.upenn.edu%2Fedissertations%2F547&utm_medium=PDF&utm_campaign=PDFCoverPages.

[5] Abdulkareem, Ademola. Oguntosin, Victoria. Popoola, Olawale M. Idowu, Ademola A. “Modeling and Nonlinear Control of a Quadcopter for Stabilization and Trajectory Tracking.” *Hindawi Journal of Engineering*, vol. 2022, 10 Oct. 2022, https://onlinelibrary.wiley.com/doi/10.1155/2022/2449901.

[6] Islam, Maidul. Okasha, Mohamed. Idres, Moumen Mohammed. “Trajectory Tracking in Quadrotor Platform by using PD Controller and LQR Approach.” *6th International Conference on Mechatronics*, 8-9 Aug. 2017, Kuala Lumpur, Malaysia, https://iopscience.iop.org/article/10.1088/1757-899X/260/1/012026.

[7] Velagic, Jasmin. Osmic, Nedium. Lacevic, Halil. “Design of LQR Controller for 3D Trajectory Tracking of Octocopter Unmanned Aerial Vehicle.” *2022 8th International Conference on Control, Decision and Information Technologies*, 17-20 May 2022, Istanbul, Turkey, https://ieeexplore.ieee.org/document/9803884.

[8] Nekoo, Saeed Rafee. Ollero, Anibal. “Experimental backward integration for state-dependent differential Riccati equation (SDDRE): A case study on flapping-wing flying robot.” *Control Engineering Practice*, vol. 151, 30 Jul. 2024, https://www.sciencedirect.com/science/article/pii/S0967066124001953?via%3Dihub.

[9] Martins, Luis. Cardeira, Carlos. Oliveira, Paulo. “Linear Quadratic Regulator for Trajectory Tracking of a Quadrotor.” *Internation federation of Automatic Control – Papers Online*, vol. 52, no. 12, 2019, pp. 176-181, https://www.sciencedirect.com/science/article/pii/S2405896319311450.

[10] Ganga, G. Dharmana, Meher Madhu. “MPC Controller for Trajectory Tracking Control of Quadcopter.” *2017 International Conference on circuits Power and Computing Technologies (ICCPCT)*, Kollam, India, 2017, pp. 1-6, https://ieeexplore.ieee.org/document/8074380.

[11] Abougarair, Ahmed J. Almograbi, Mohamed. Alaktiw, Abdusalam A. “Model Predictive Control for Stabilizing Quadcopter Flight and Following Trajectories.” *The Internation Journal of Engineering and Information Technology (IJEIT)*, vol. 13, no. 2, Jun. 2025, https://www.researchgate.net/publication/391021822_Model_Predictive_Control_for_Stabilizing_Quadcopter_Flight_and_Following_Trajectories.

[12] Islam, Maidul. Okasha, Mohamed. Idres, Moumen Mohammed. “ Dynamics and Control of Quadcopter using Linear Model Predictive Control Approach.” AEROS Conference, 12 Dec. 2017, Putrajaya, Malaysia, https://iopscience.iop.org/article/10.1088/1757-899X/270/1/012007.

[13] Varma, Bhaskar. Swamy, Nitin. Mukherjee, Sujoy. “Trajectory Tracking of Autonomous Vehicles using Different Control Techniques (PID vs LQR vs MPC).” *International Conference on Smart Technologies in Computing, Electrical and Electronics (ICSTCEE 2020)*, Bengaluru, India, 2020, pp. 84-89, https://ieeexplore.ieee.org/document/9276986.

[14] Maji, Abhishek. “*Learning Model Predictive Control with Application to Quadcopter Trajectory Tracking*.” Master in Electric Power Engineering, KTH Royal Institute of Technology, 13 Feb. 2020. DiVA Portal, https://www.diva-portal.org/smash/get/diva2:1453324/FULLTEXT01.pdf.

[15] Aldieen, Mohamed Imad. “*Design of a typical multi role vehicle using quad-rotor theory*.” Thesis for Bachelor’s Degree, Sudan University of Science and Technology, Aug. 2014. Research Gate, https://www.researchgate.net/publication/303923707_Design_of_a_typical_multi_role_vehicle_using_quad-rotor_theory?_tp=eyJjb250ZXh0Ijp7ImZpcnN0UGFnZSI6InB1YmxpY2F0aW9uIiwicGFnZSI6Il9kaXJlY3QifX0.

[16] “lqr.” *MATLAB and Simulink*, MathWorks, https://www.mathworks.com/help/control/ref/lti.lqr.html.

[17] “lqi.” *MATLAB and Simulink*, MathWorks, https://www.mathworks.com/help/control/ref/ss.lqi.html.

[18] Russ Tedrake. “*Underactuated Robotics: Algorithms for Walking, Running, Swimming, Flying, and Manipulation (Course Notes for MIT 6.832)”.* 2023. Downloaded on 30 Jul. 2025 from https://underactuated.csail.mit.edu/

[19] “mpc.” *MATLAB and Simulink*, MathWorks, https://www.mathworks.com/help/mpc/ref/mpc.html.

[20] “Model Predictive Control Toolbox.” *MATLAB and Simulink*, MathWorks, https://www.mathworks.com/help/mpc/ref/mpc.htmlhttps://www.mathworks.com/help/mpc/ug/optimization-problem.html.

[21] Åström, Karl J., and Tore Hägglund. *Advanced PID Control*. 2nd ed., ISA – The Instrumentation, Systems, and Automation Society, 2006.

[22] “Genetic Algorithm.” *Cornell University Computational Optimization Open Textbook – Optimization Wiki*, Cornell University, https://optimization.cbe.cornell.edu/index.php?title=Genetic_algorithm.

[23] “Comparing Genetic Algorithm and Particle Swarm Optimization in Optimization Problems.” *Science of Biogenetics*, 20 Dec. 2023, https://scienceofbiogenetics.com/articles/comparing-genetic-algorithm-and-particle-swarm-optimization-in-optimization-problems.

[24] “ga.” *MATLAB and Simulink*, MathWorks, https://www.mathworks.com/help/gads/ga.html.

[25] “Autoregressive (AR) Models.” Learning TS, https://sjmiller8182.github.io/LearningTS/build/autoregressive_models.html.

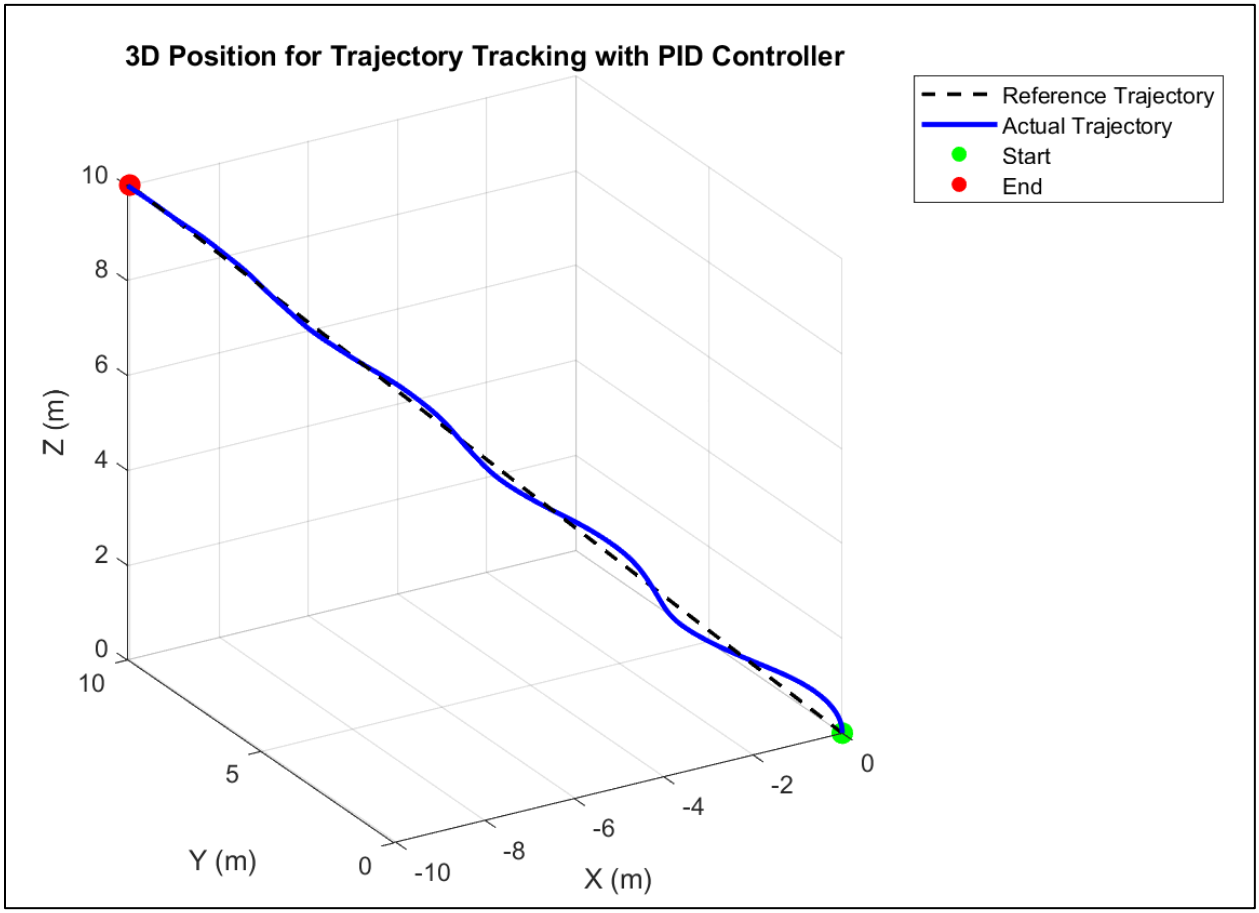
[26] “Calculus III – 3-Dimensional Space – Curvature.” Paul’s Online Notes, Lamar University, 16 Nov. 2022, https://tutorial.math.lamar.edu/Classes/CalcIII/Curvature.aspx.

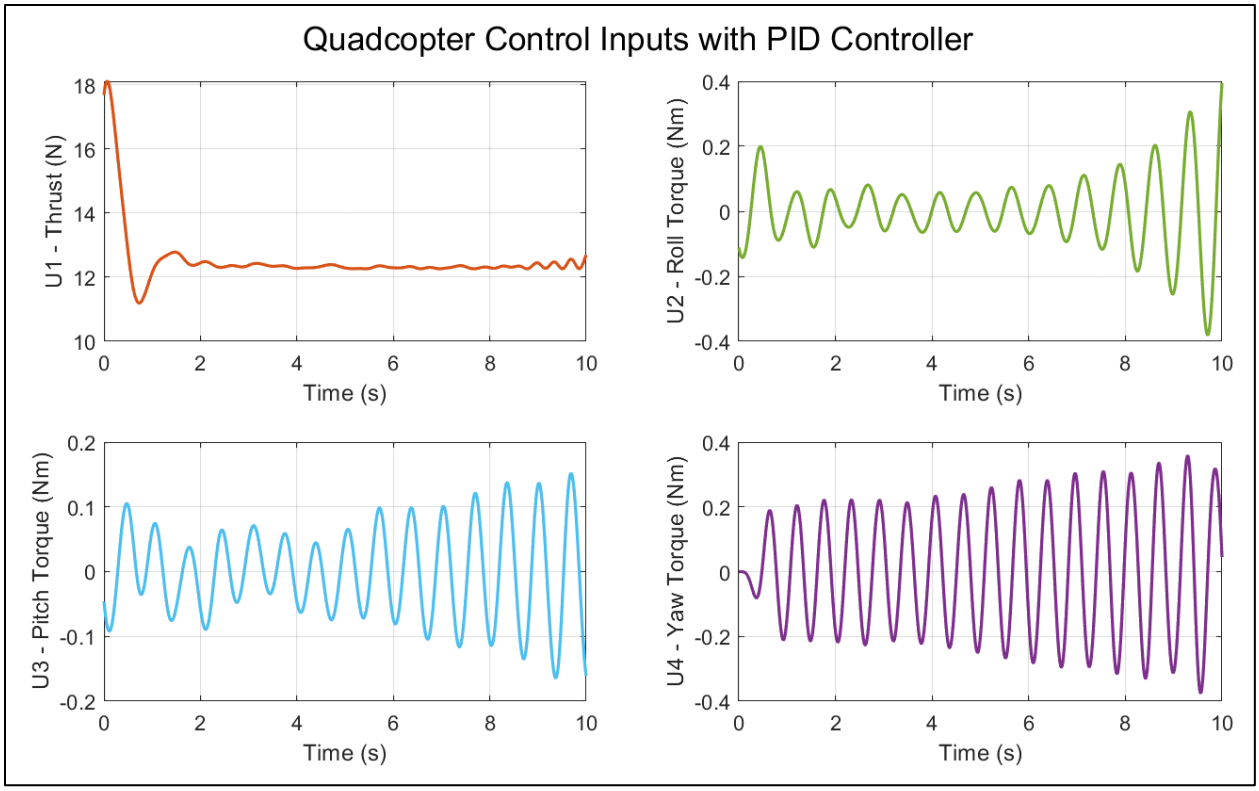
[27] “findpeaks.” *MATLAB and Simulink*, MathWorks, https://www.mathworks.com/help/signal/ref/findpeaks.html.

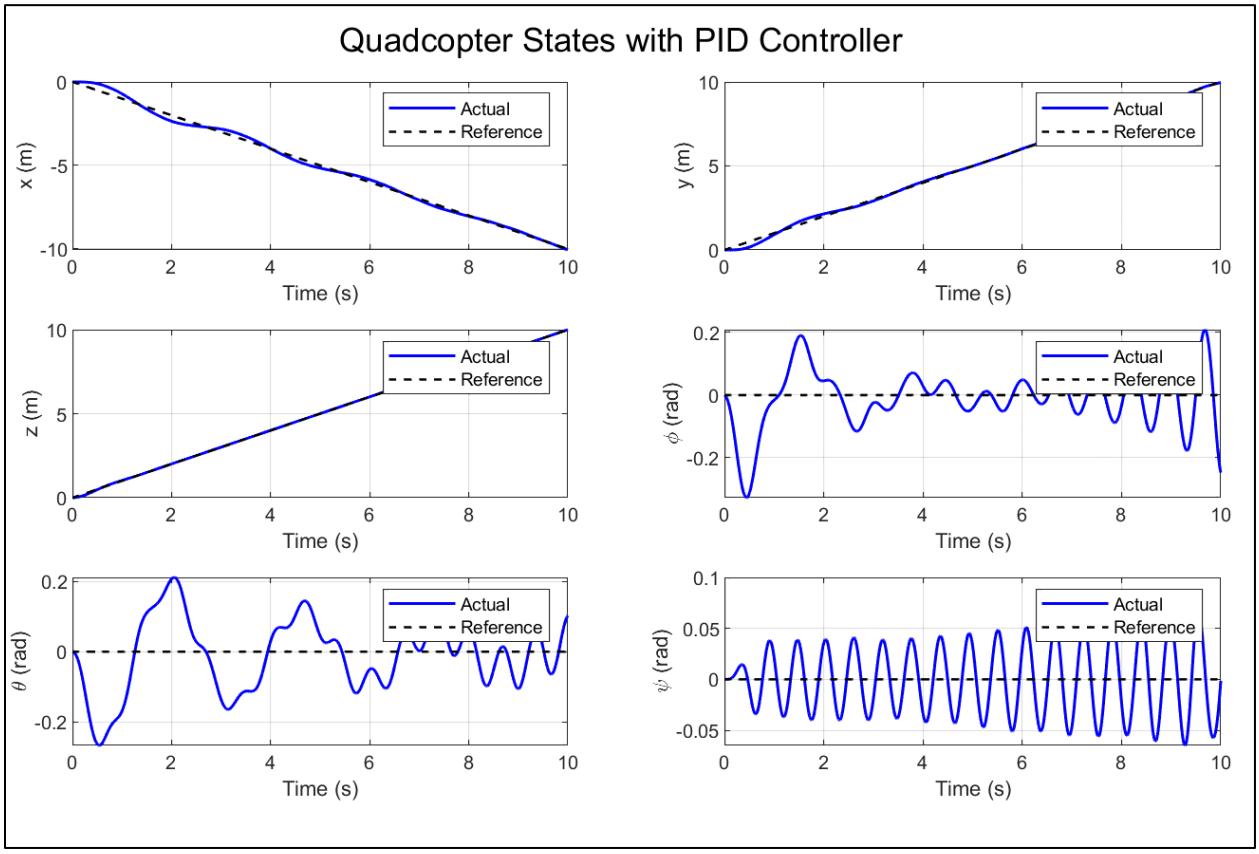
Appendix A

PID Controller Performance Plots

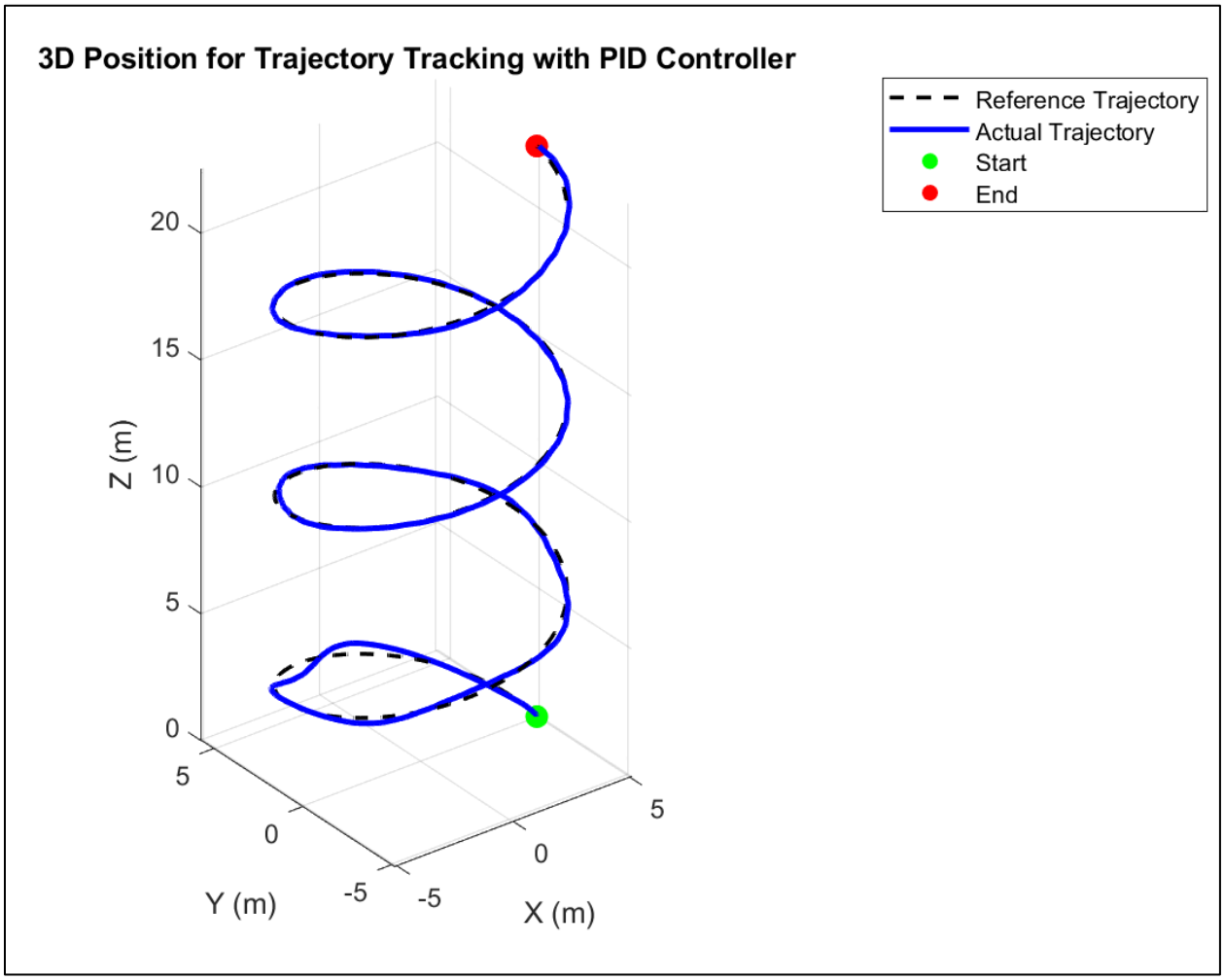
3D Line Trajectory

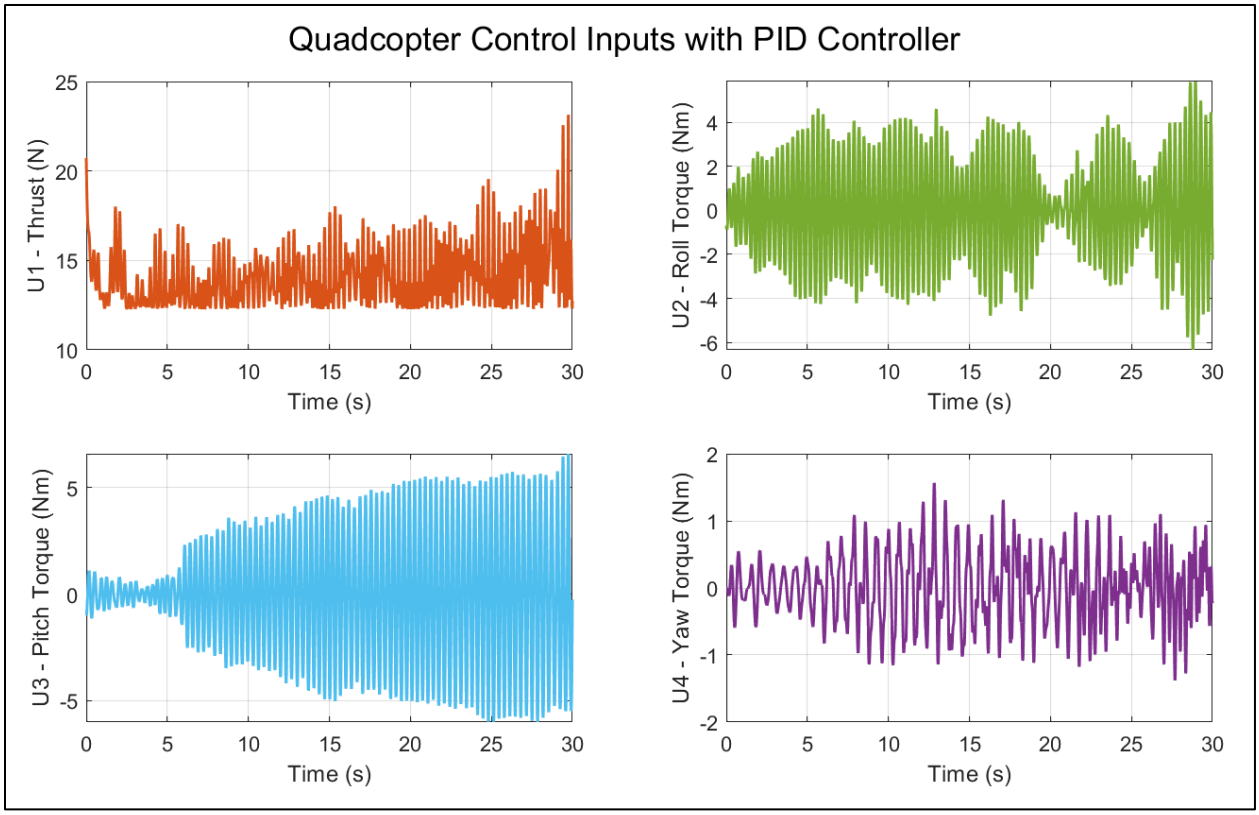


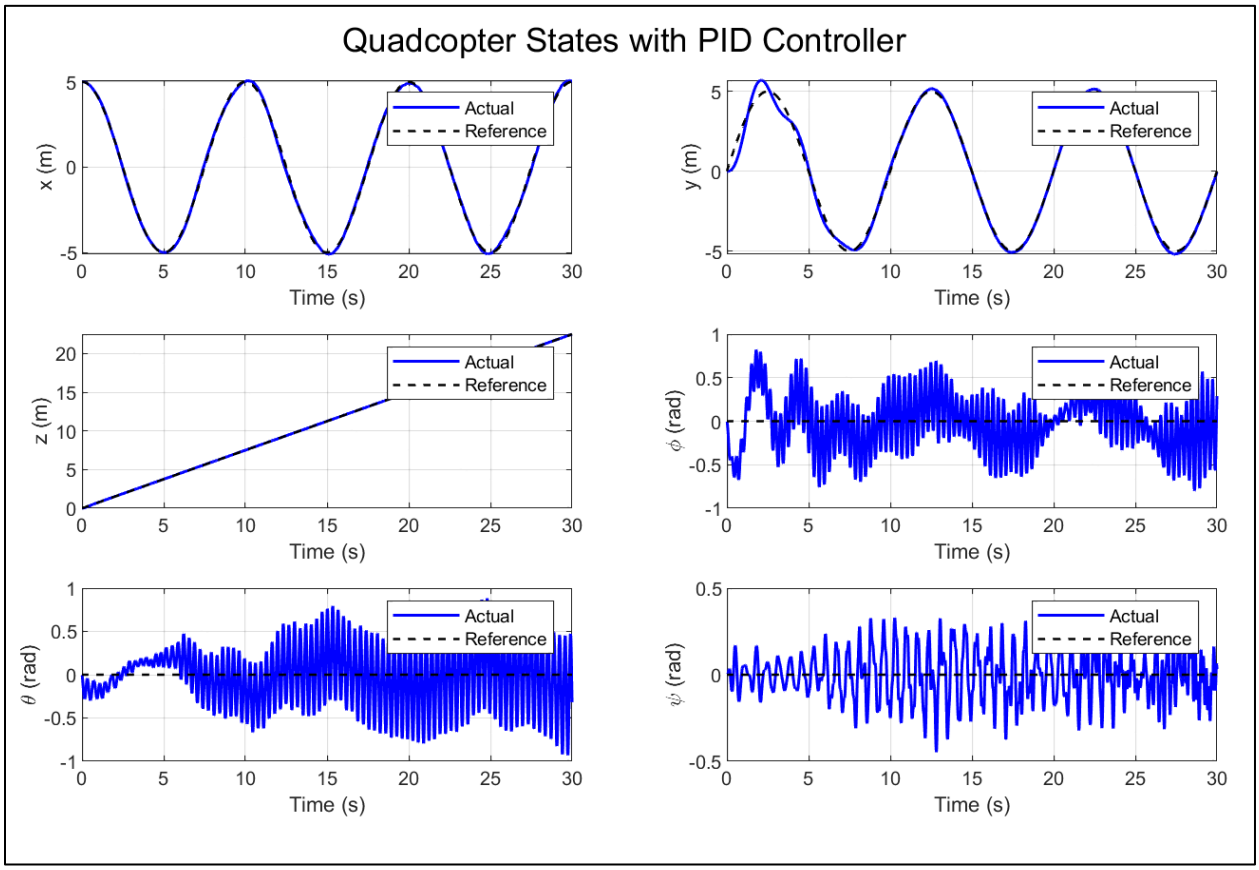

Quadcopter Control Inputs with PID Controller
U1 - Thrust (N)
U2 - Roll Torque (Nm)
U3 - Pitch Torque (Nm)
U4 - Yaw Torque (Nm)
Time (s)


Quadcopter States with PID Controller
x (m)
y (m)
z (m)
ϕ (rad)
θ (rad)
ψ (rad)
Actual
Reference
Time (s)

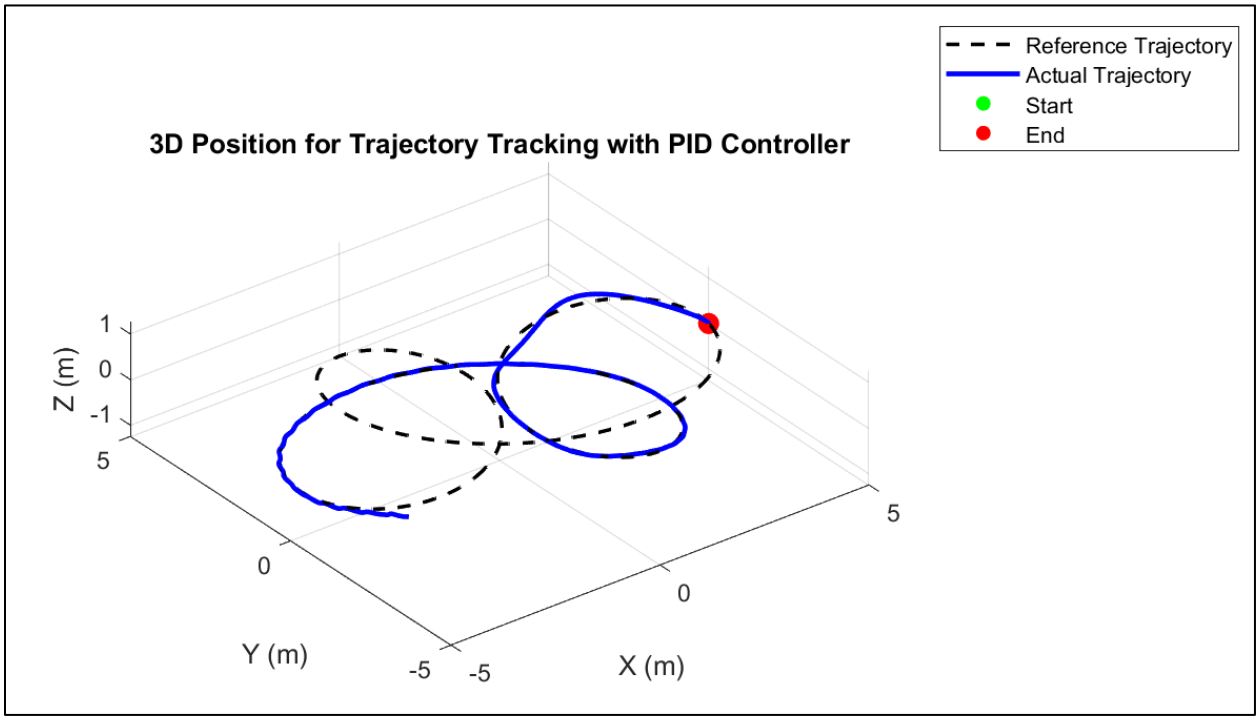
Upwards Spiral Trajectory

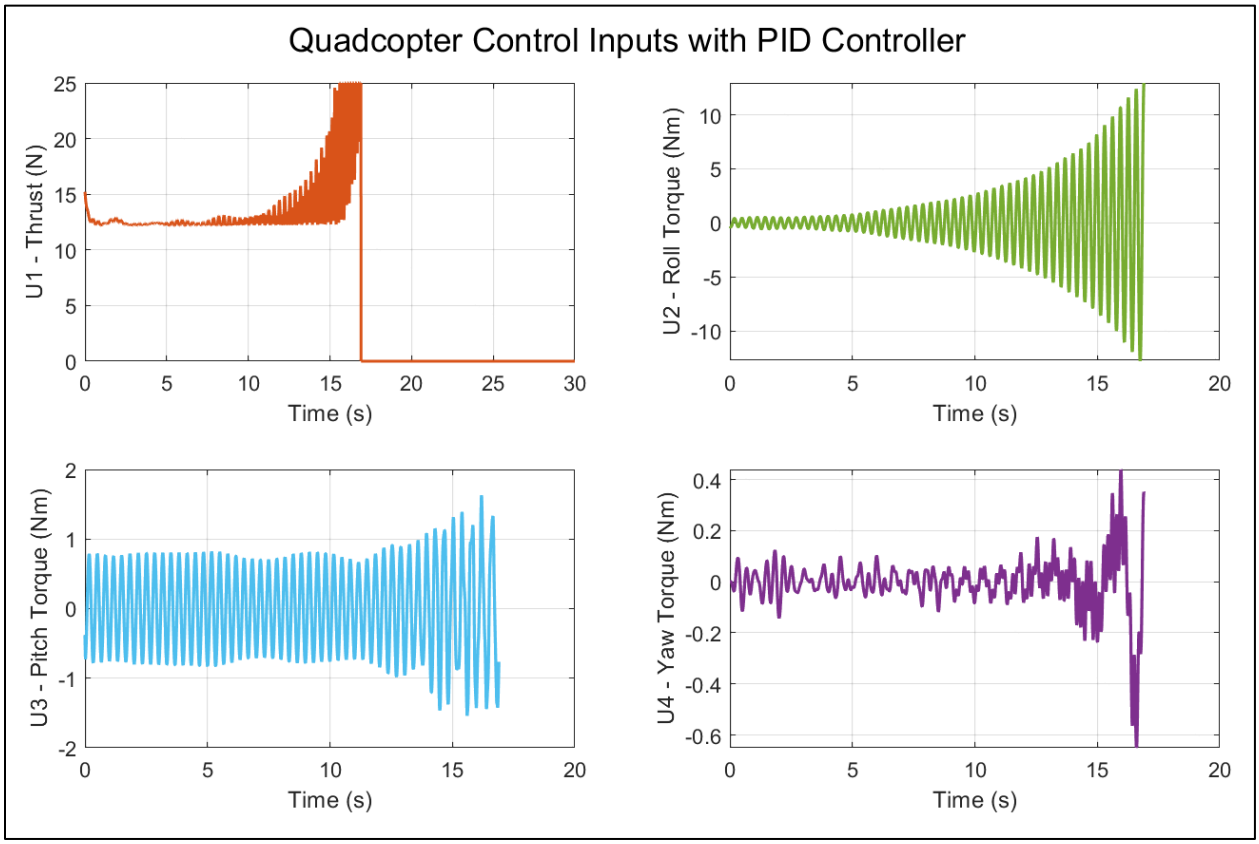


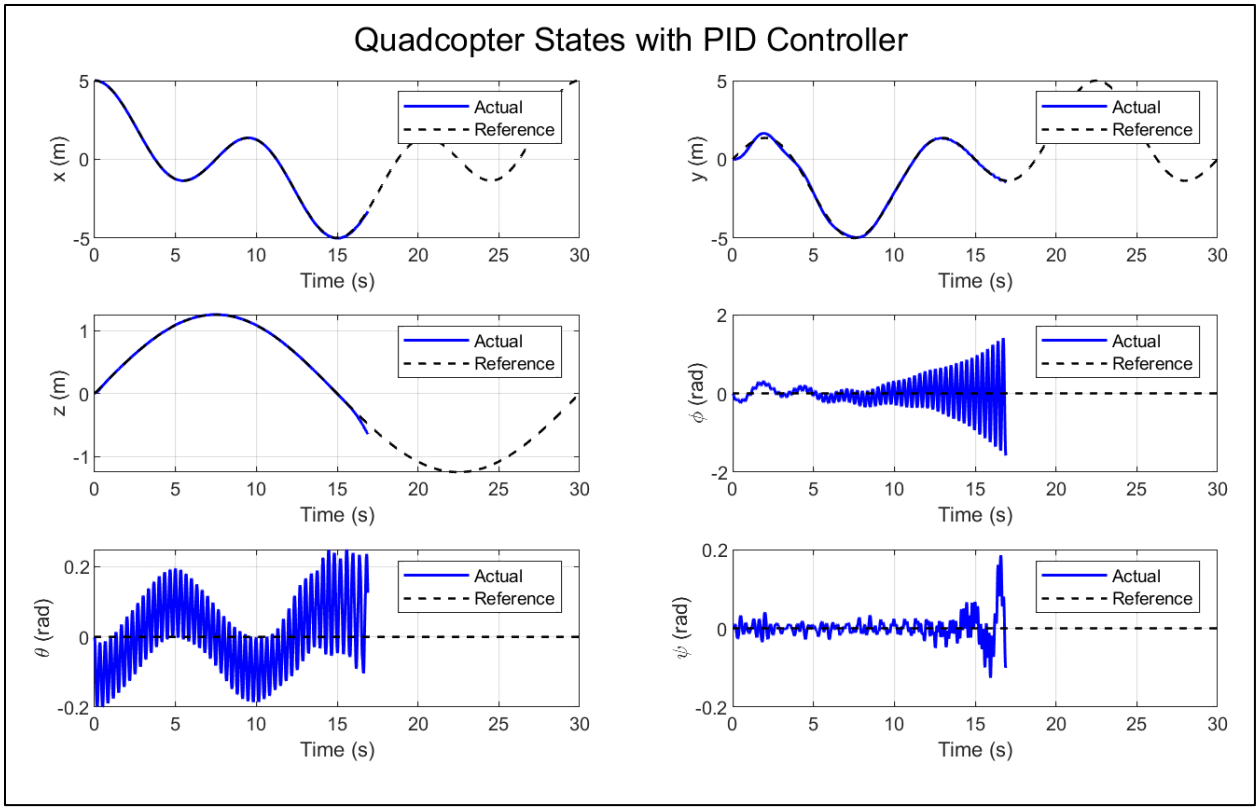

Quadcopter Control Inputs with PID Controller
U1 - Thrust (N)
U2 - Roll Torque (Nm)
U3 - Pitch Torque (Nm)
U4 - Yaw Torque (Nm)
Time (s)


Quadcopter States with PID Controller
x (m)
y (m)
z (m)
ϕ (rad)
θ (rad)
ψ (rad)
Actual
Reference
Time (s)

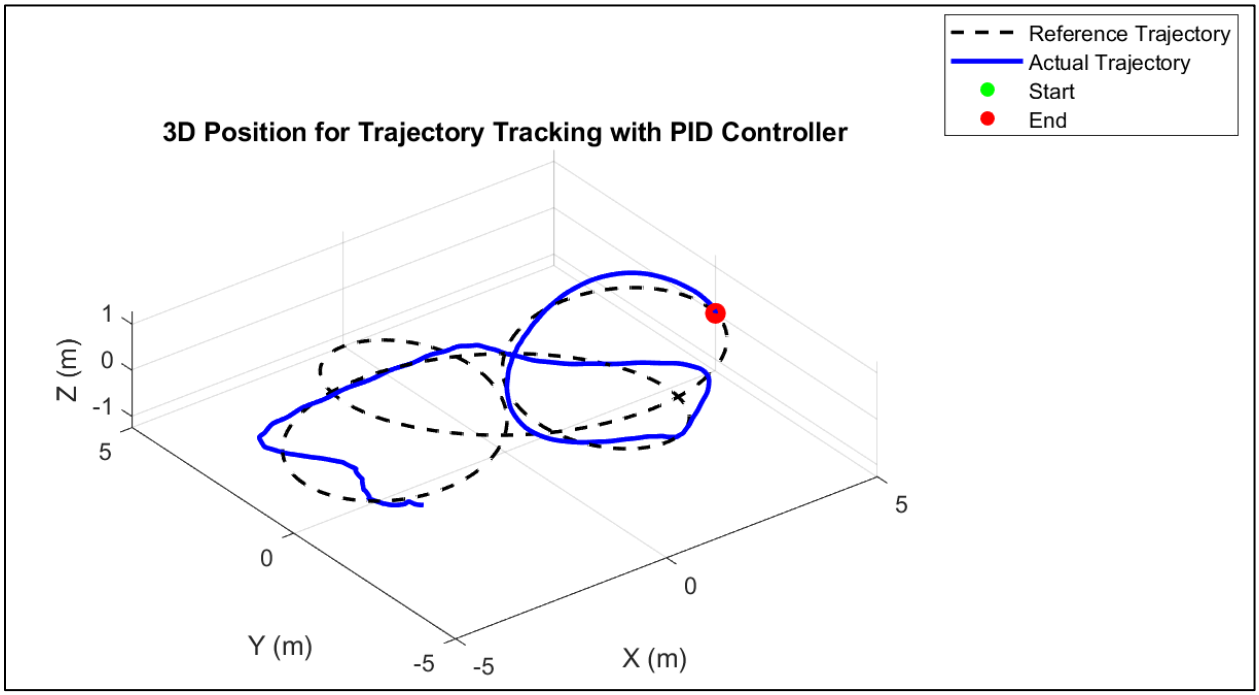
Rose Petal Curve Trajectory

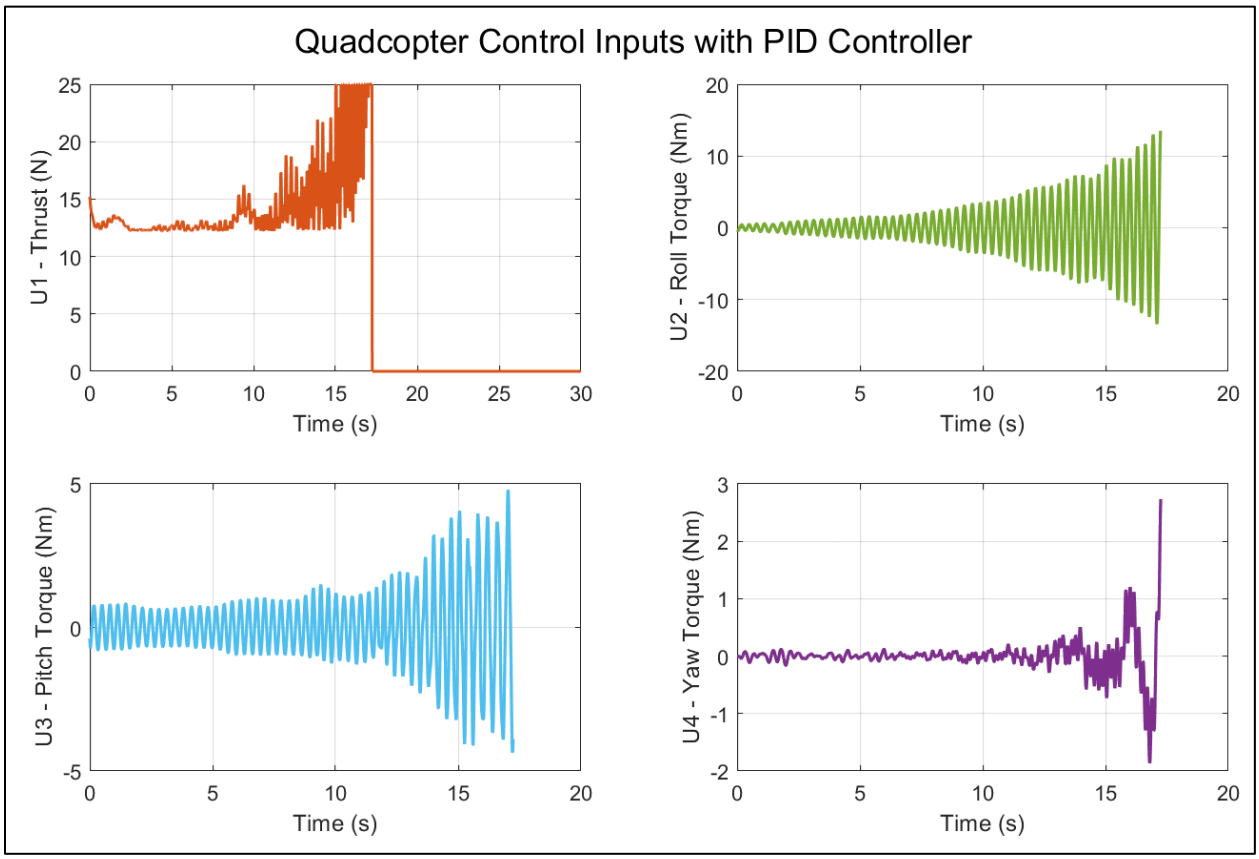


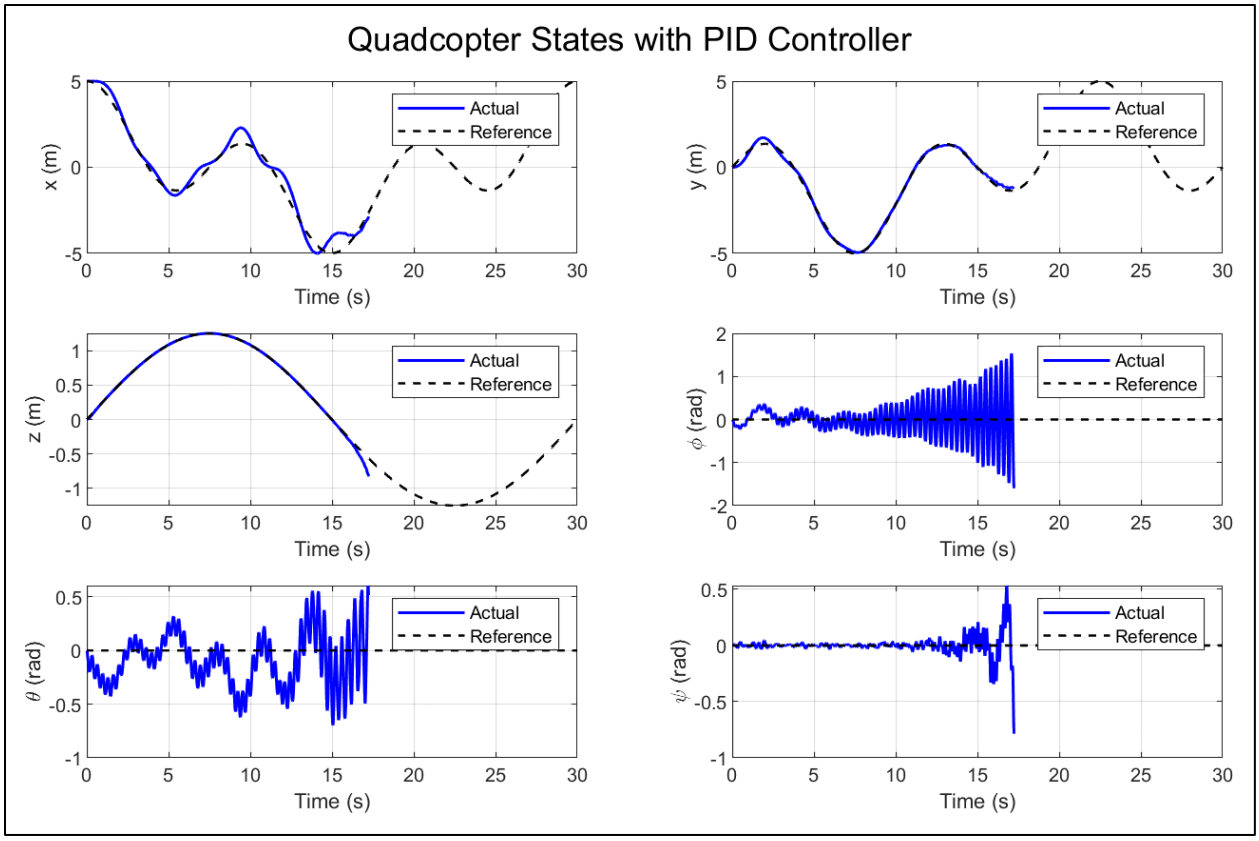



Quadcopter States with PID Controller
x (m)
y (m)
z (m)
ϕ (rad)
θ (rad)
ψ (rad)
Time (s)
Actual
Reference

Rose Petal Curve Trajectory with Wind Disturbance

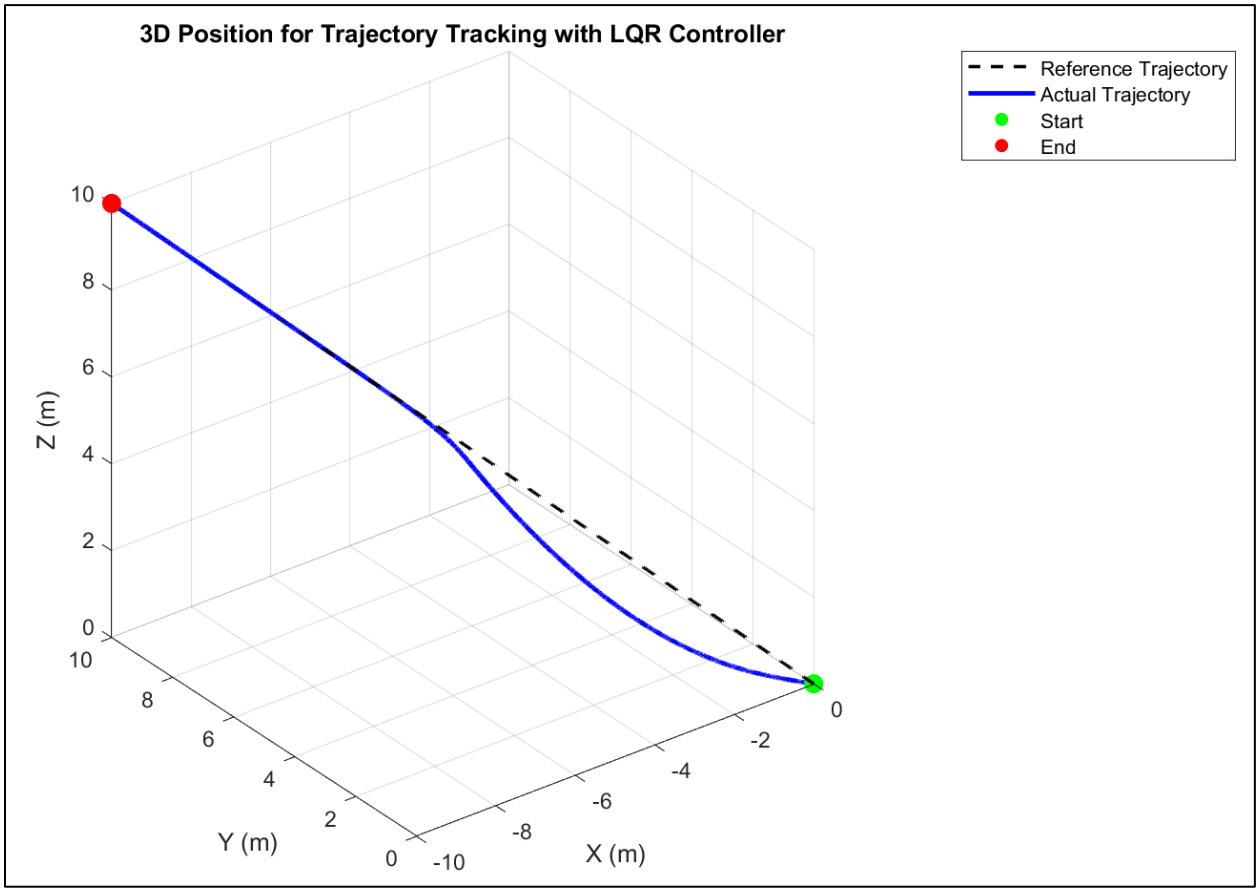


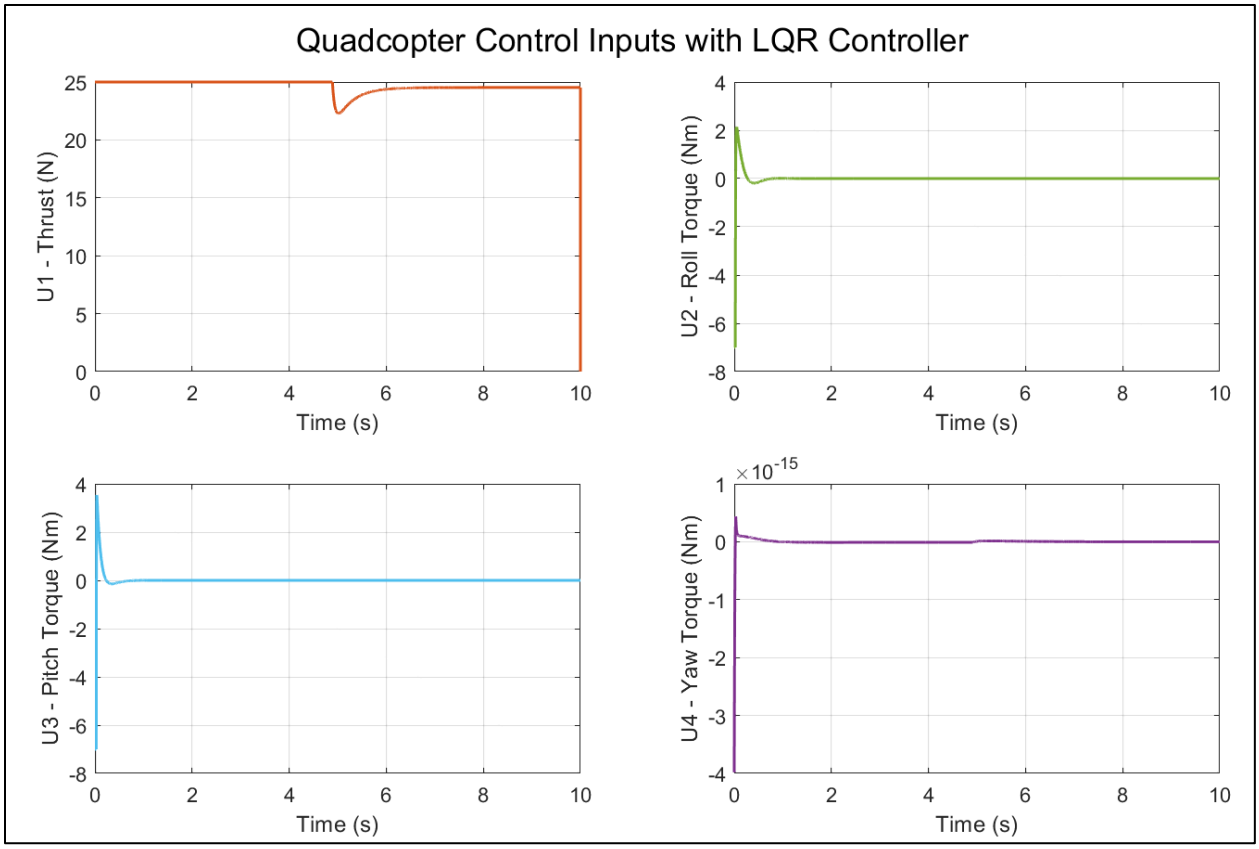


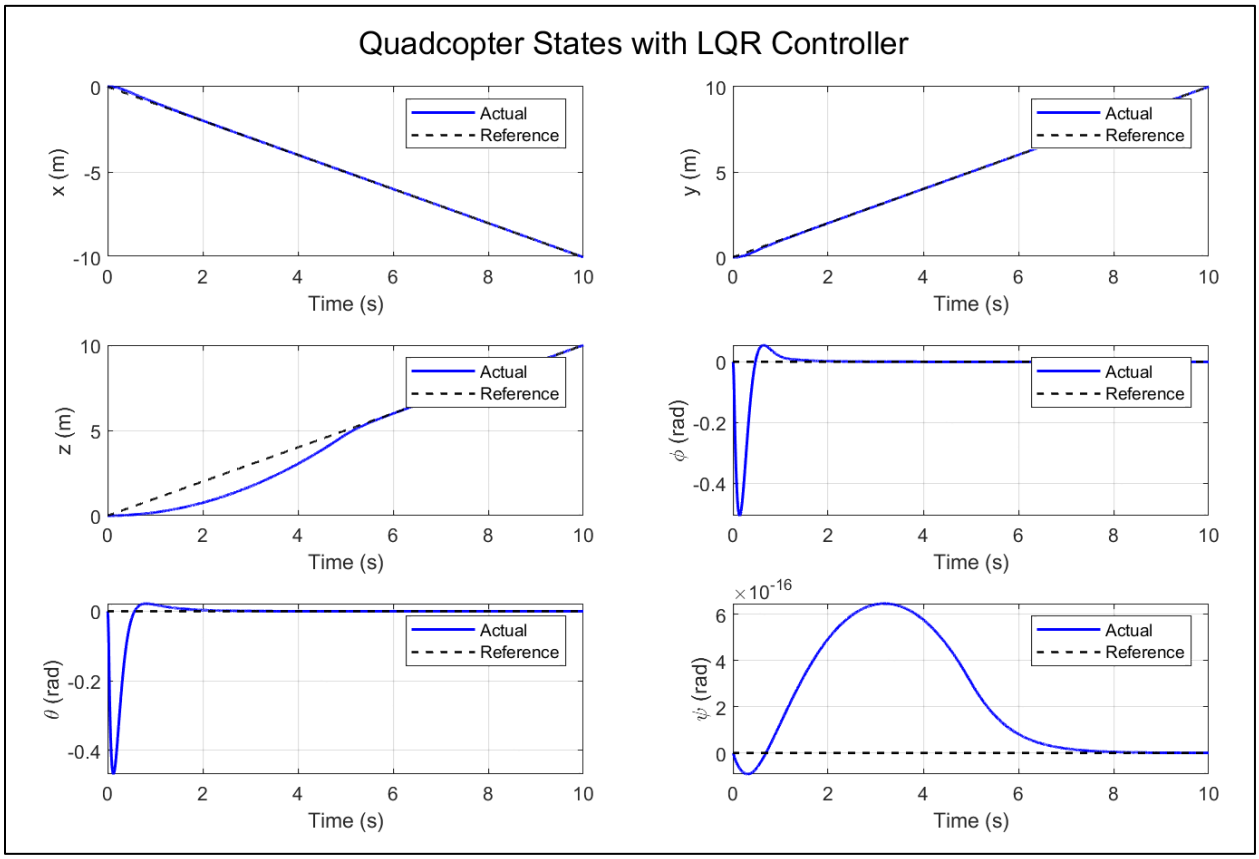

Quadcopter States with PID Controller
x (m)
y (m)
z (m)
ϕ (rad)
θ (rad)
ψ (rad)
Time (s)
Actual
Reference

LQR Controller Performance Plots

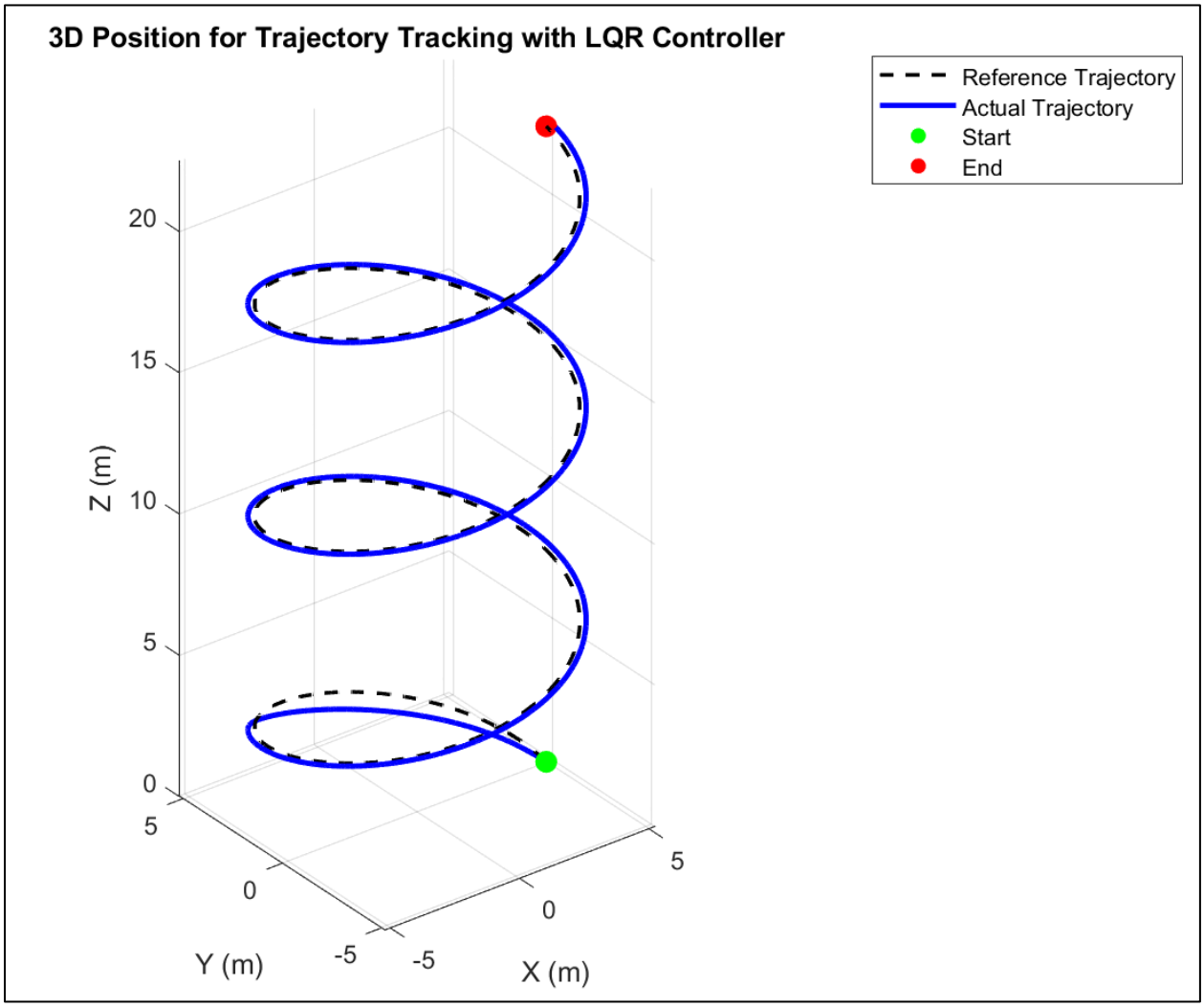
3D Line Trajectory

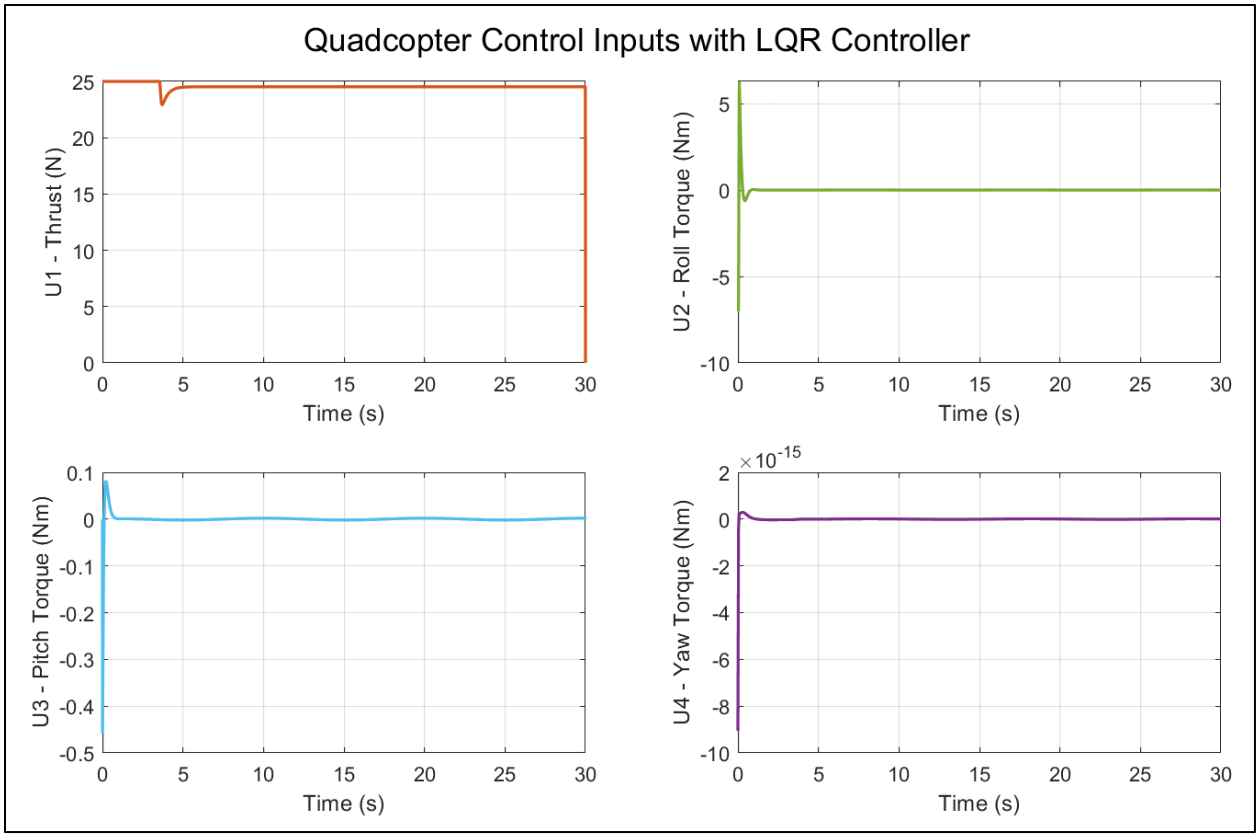


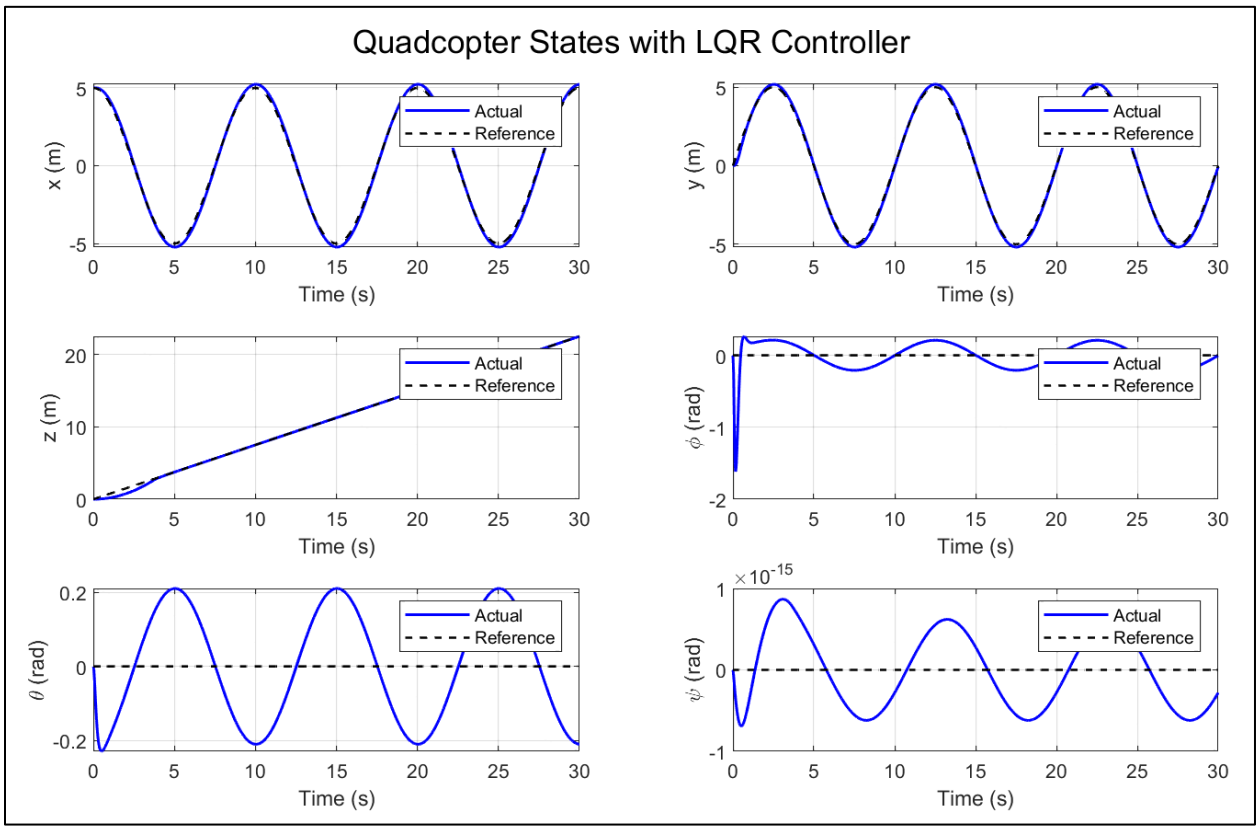

Quadcopter Control Inputs with LQR Controller
U1 - Thrust (N)
U2 - Roll Torque (Nm)
U3 - Pitch Torque (Nm)
U4 - Yaw Torque (Nm)
×10^-15
Time (s)


Quadcopter States with LQR Controller
x (m)
y (m)
z (m)
ϕ (rad)
θ (rad)
ψ (rad)
×10^-16
Actual
Reference
Time (s)

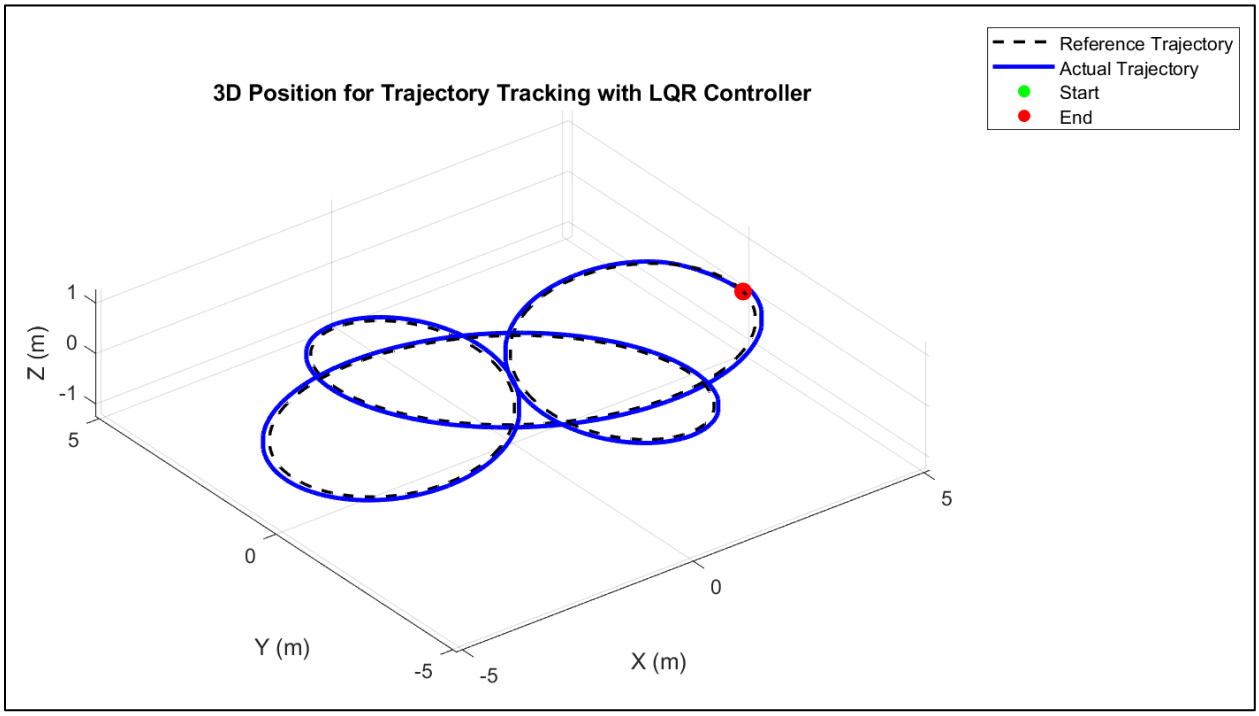
Upwards Spiral Trajectory

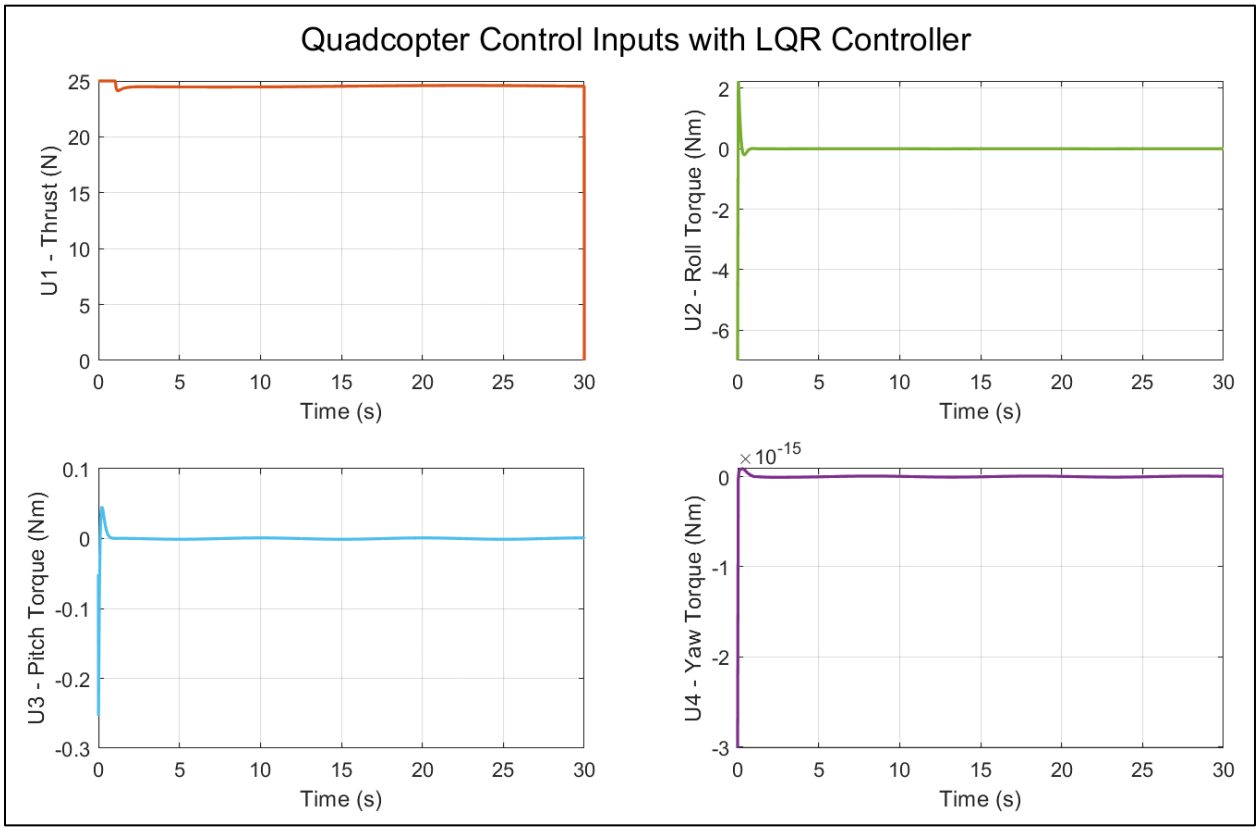


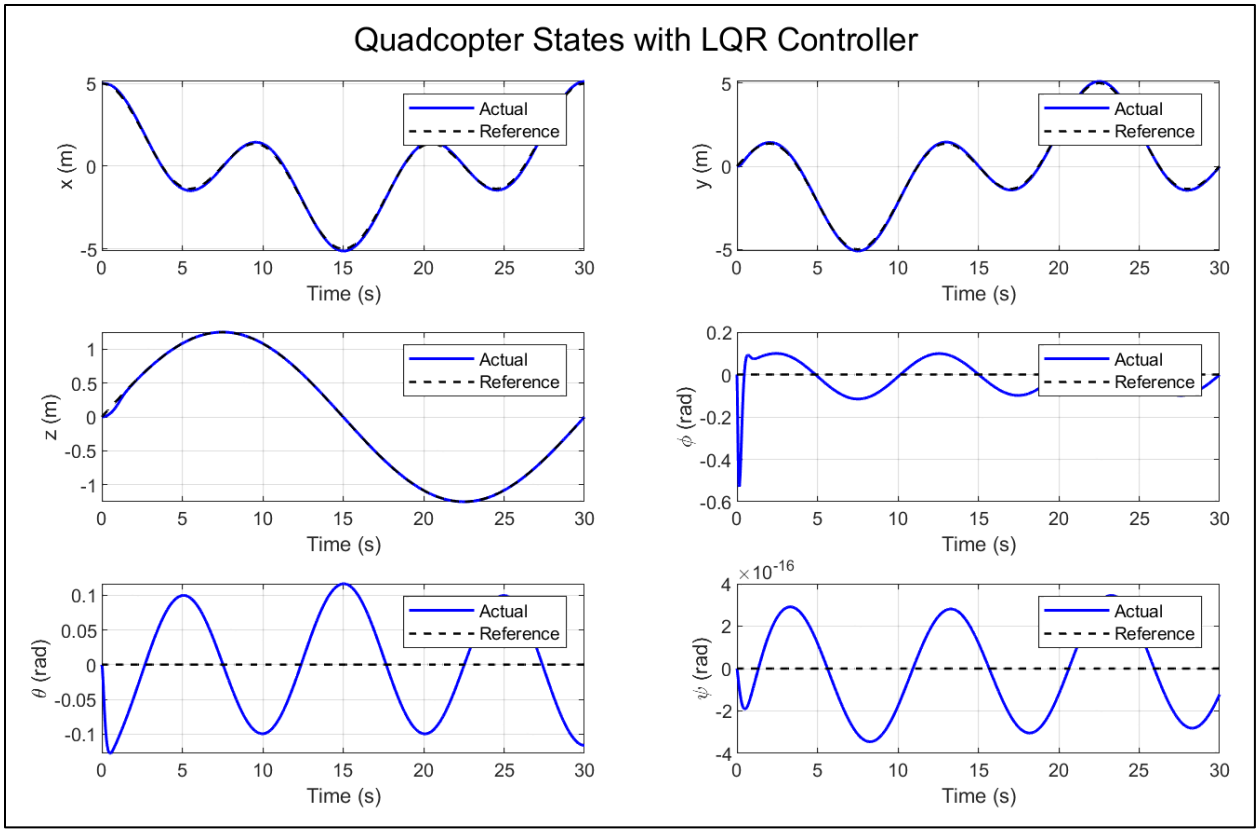

Quadcopter Control Inputs with LQR Controller
U1 - Thrust (N)
U2 - Roll Torque (Nm)
U3 - Pitch Torque (Nm)
U4 - Yaw Torque (Nm)
Time (s)
×10^-15


Quadcopter States with LQR Controller
x (m)
y (m)
z (m)
ϕ (rad)
θ (rad)
ψ (rad)
Actual
Reference
Time (s)
×10^-15

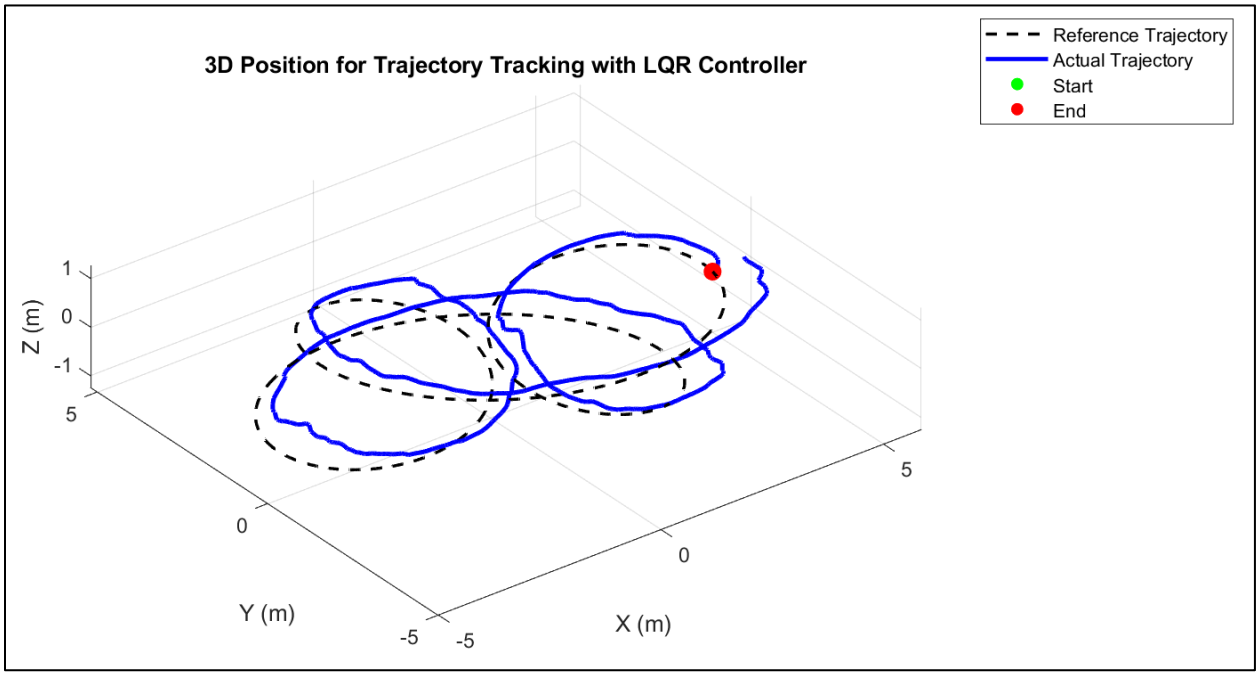
Rose Petal Curve Trajectory

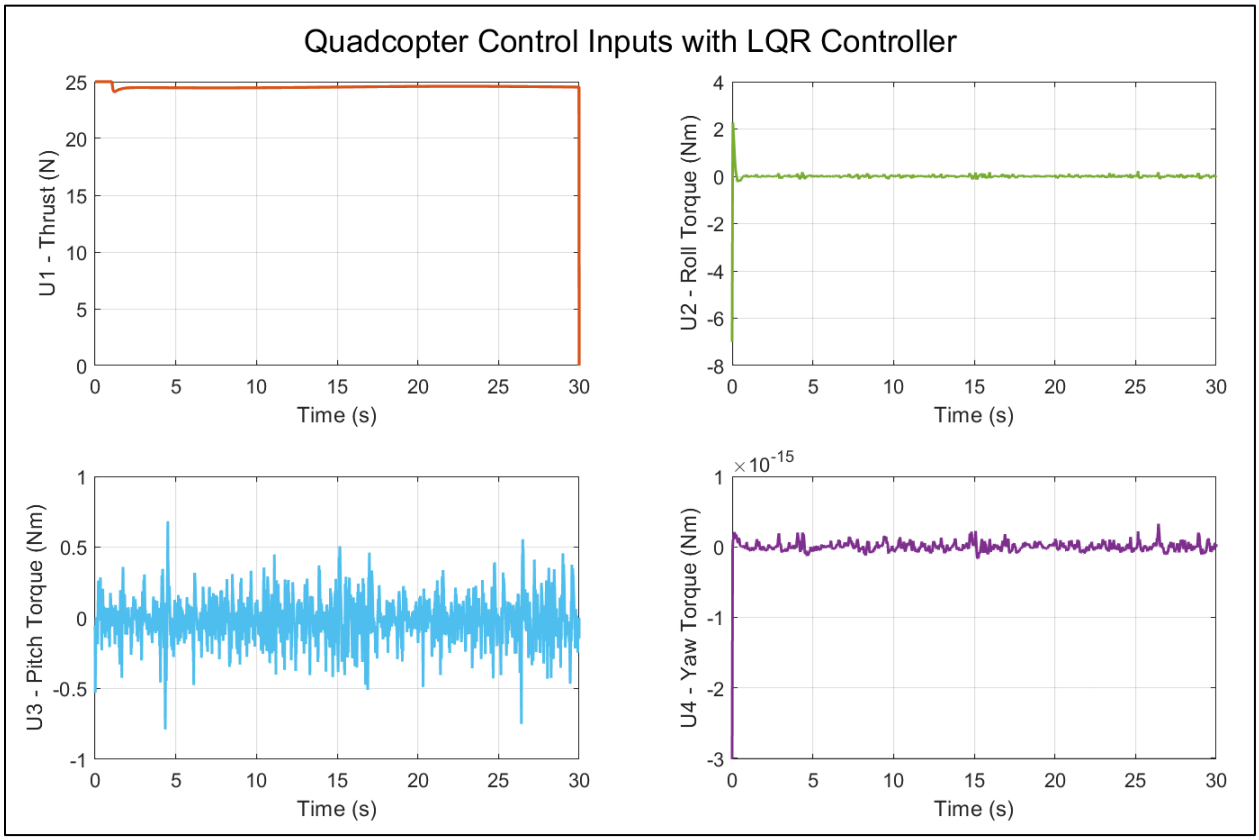


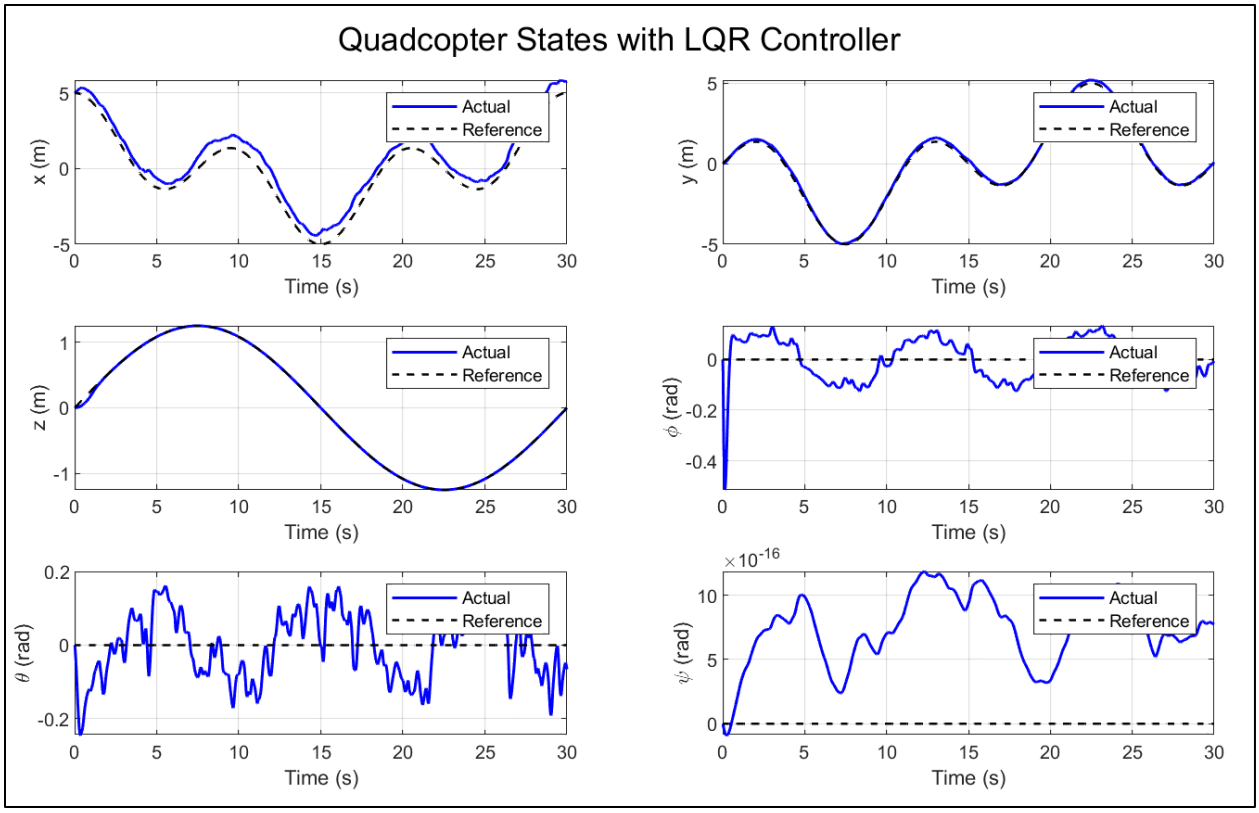



Quadcopter States with LQR Controller
x (m)
y (m)
z (m)
ϕ (rad)
θ (rad)
ψ (rad)
Time (s)
Actual
Reference
$\times 10^{-16}$

Rose Petal Curve Trajectory with Wind Disturbance

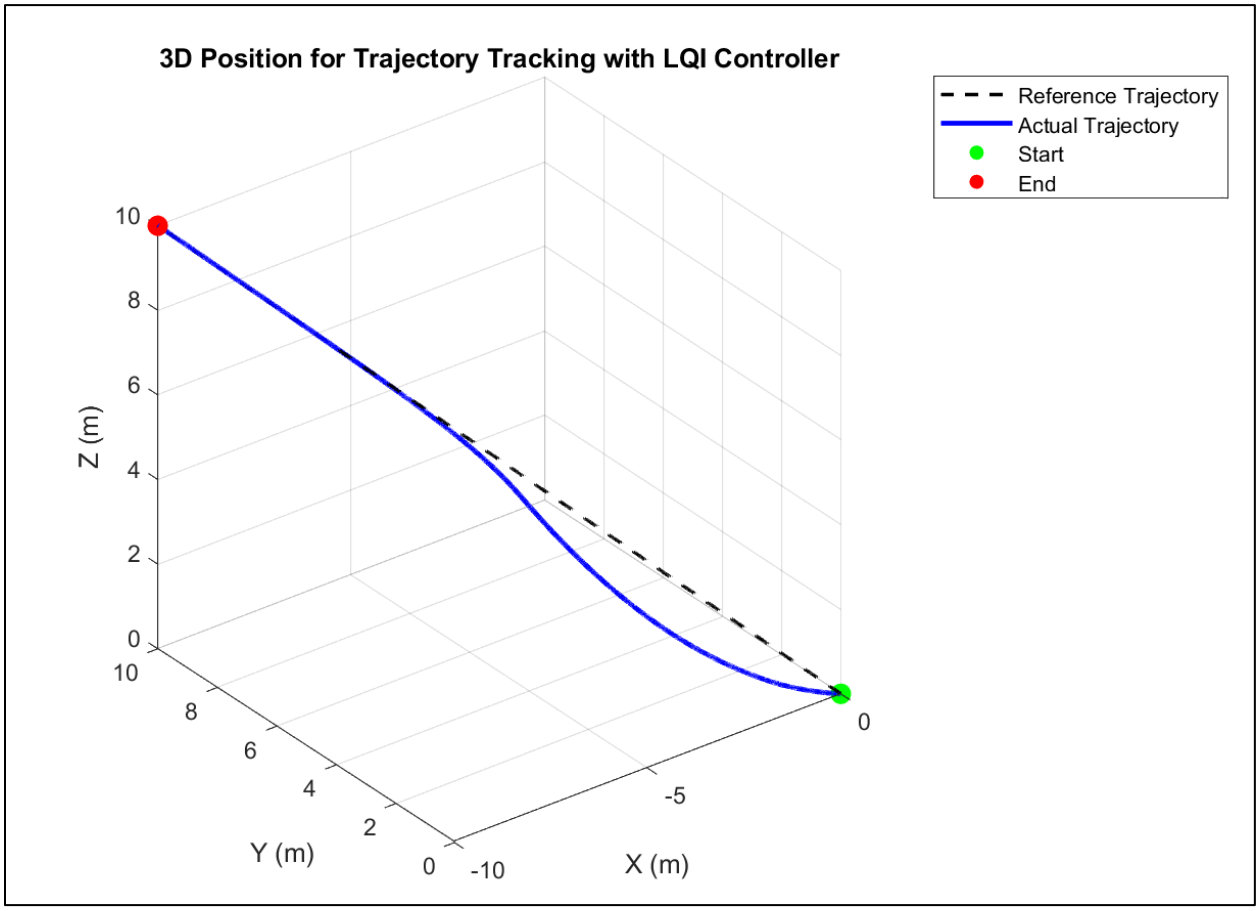


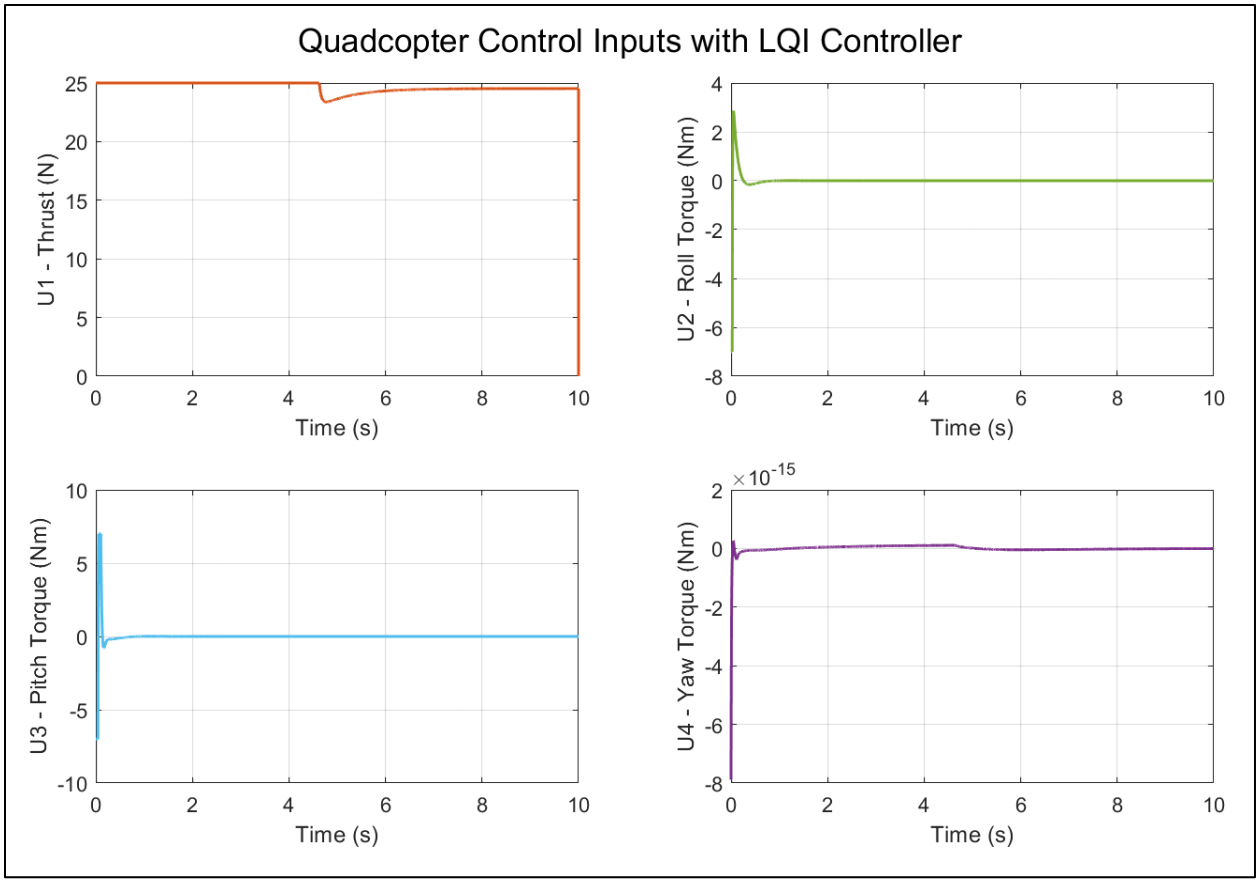


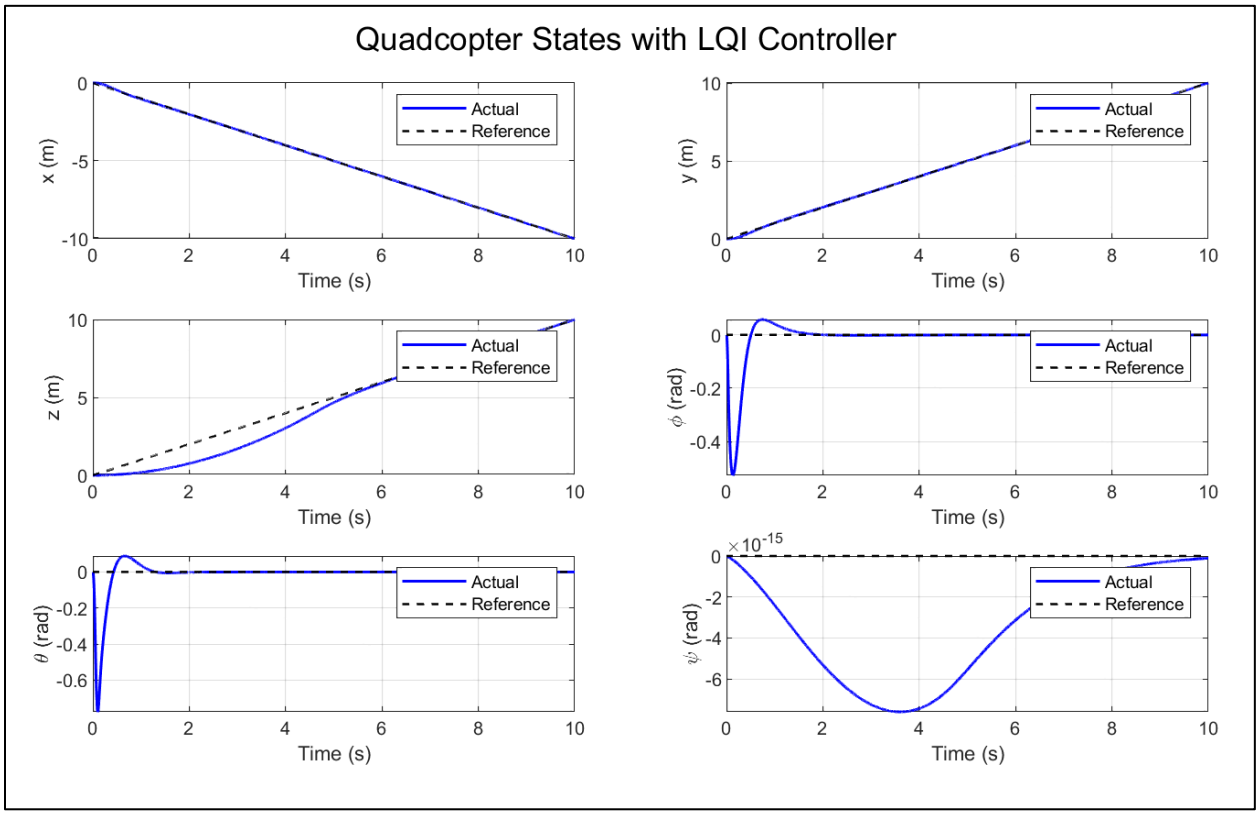

Quadcopter States with LQR Controller
x (m)
y (m)
z (m)
ϕ (rad)
θ (rad)
ψ (rad)
$\times 10^{-16}$
Time (s)
Actual
Reference

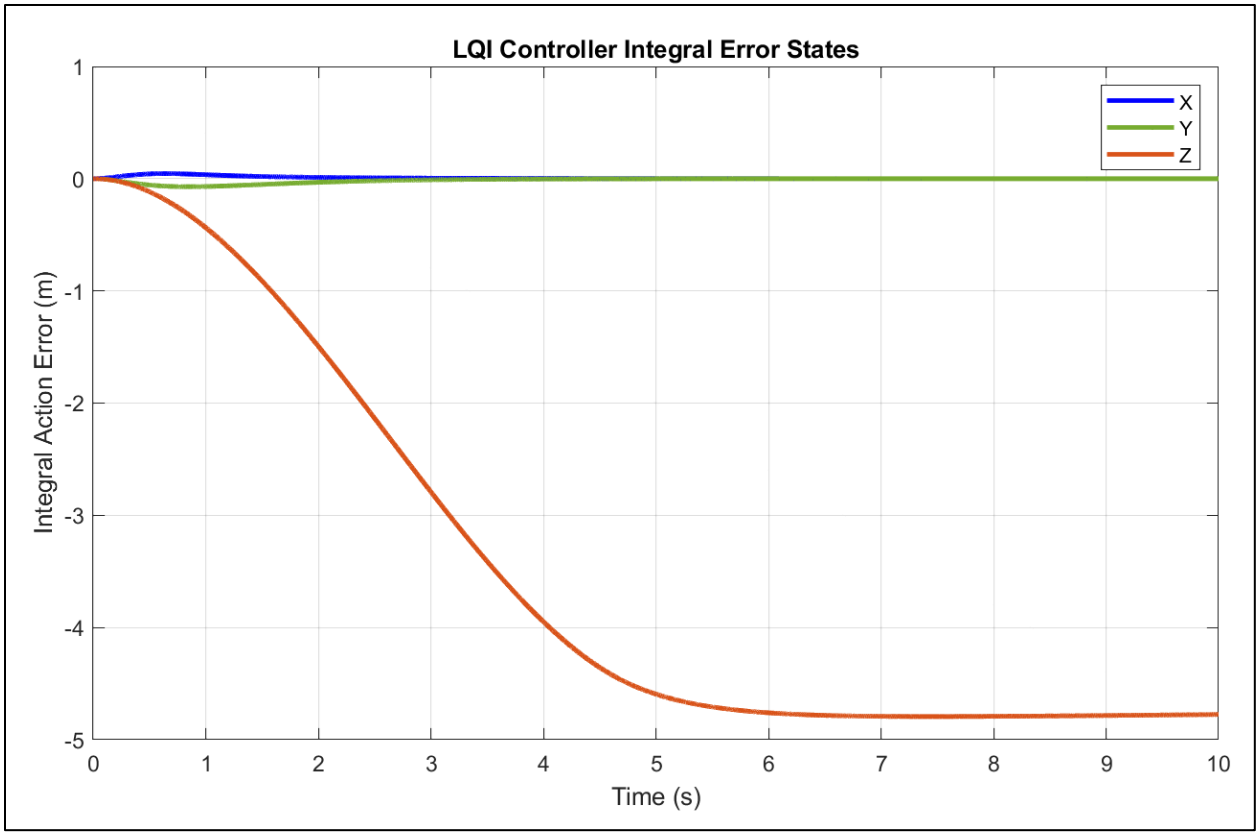
LQI Controller Performance Plots

3D Line Trajectory

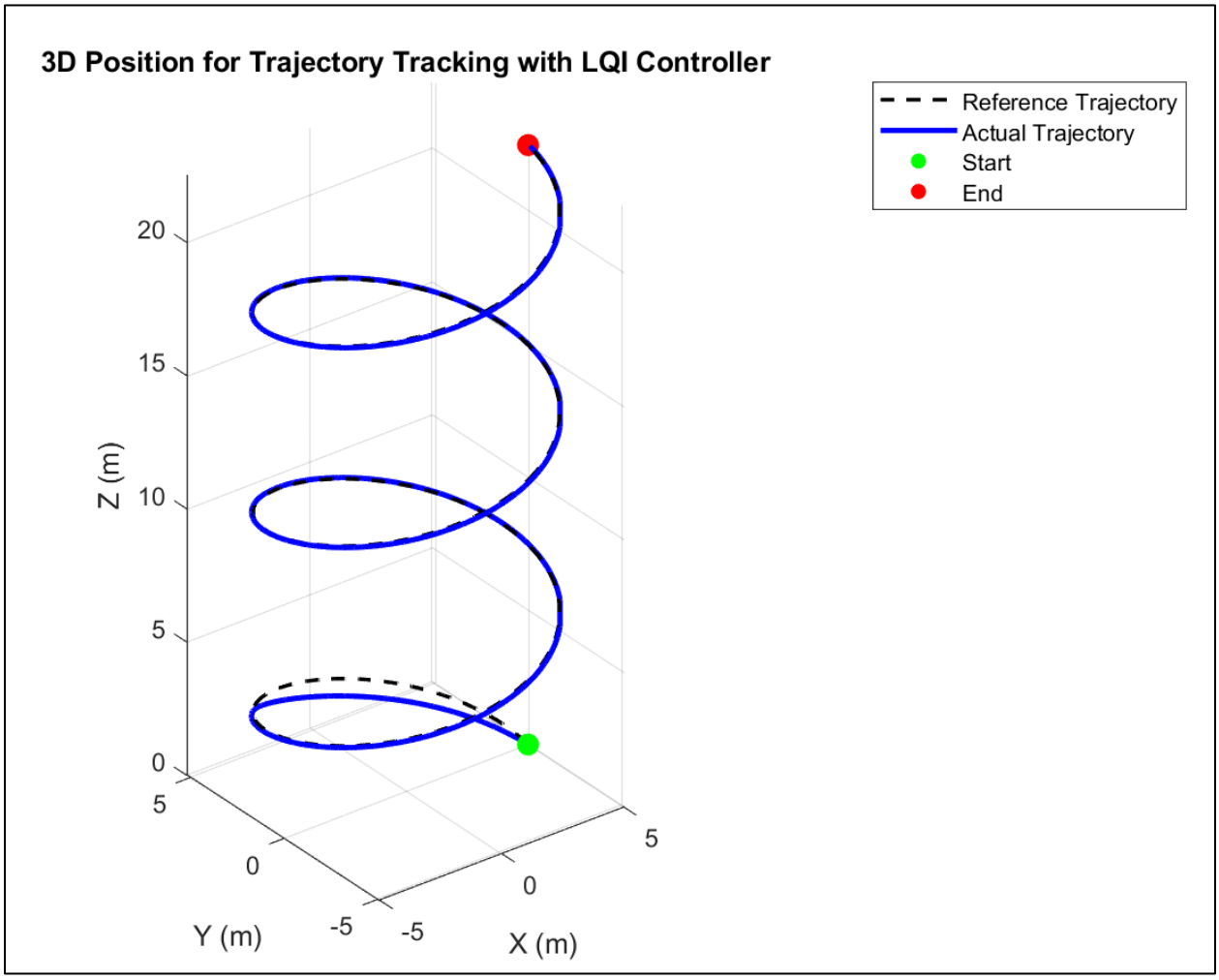


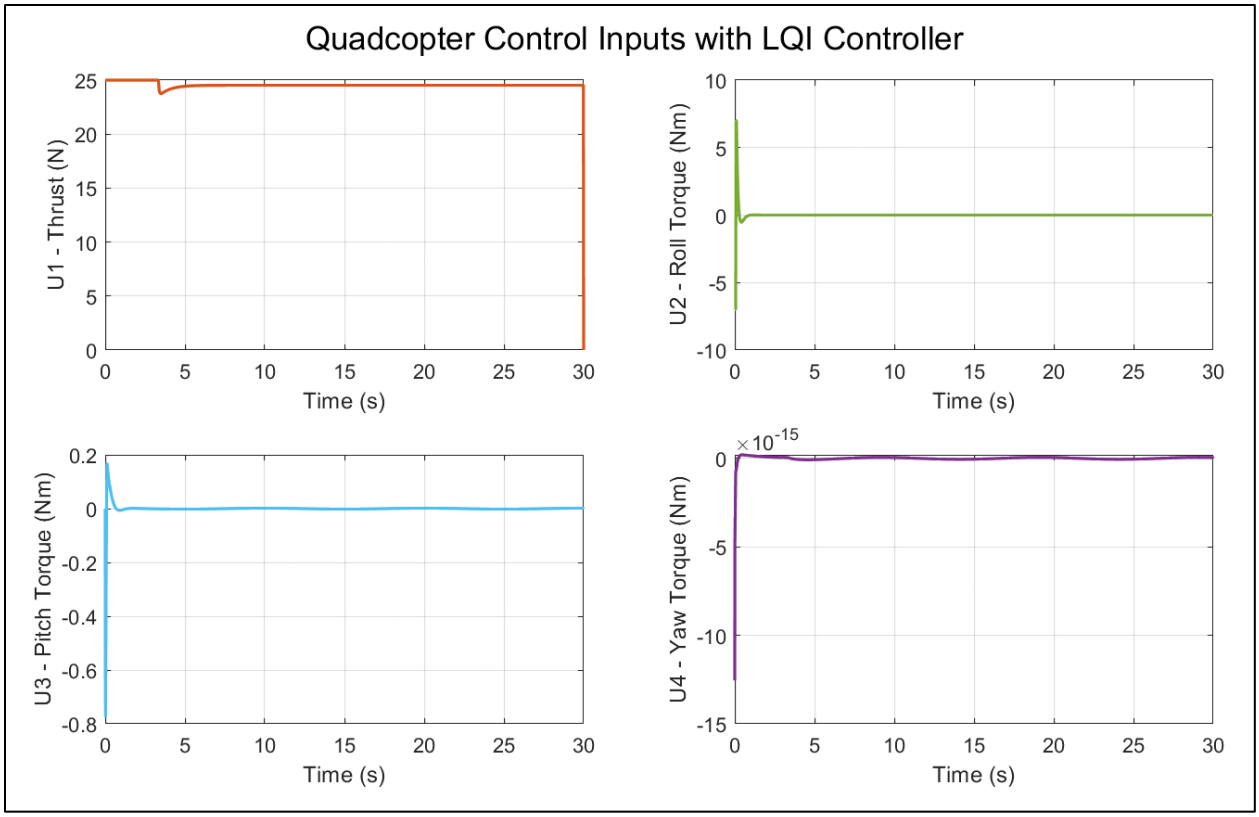

Quadcopter Control Inputs with LQI Controller
U1 - Thrust (N)
U2 - Roll Torque (Nm)
U3 - Pitch Torque (Nm)
U4 - Yaw Torque (Nm)
Time (s)
×10^-15

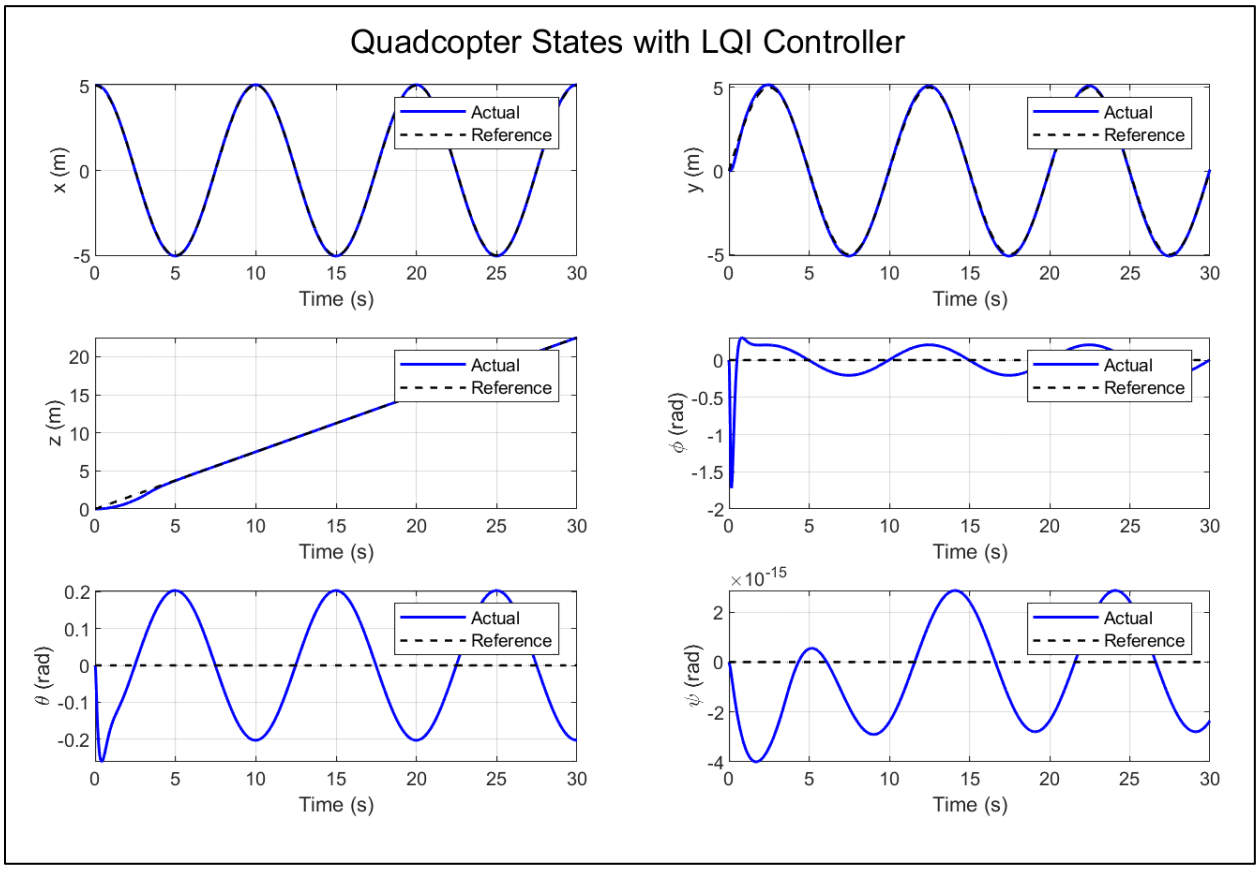

Quadcopter States with LQI Controller
x (m)
y (m)
z (m)
ϕ (rad)
θ (rad)
ψ (rad)
Actual
Reference
Time (s)
×10^-15

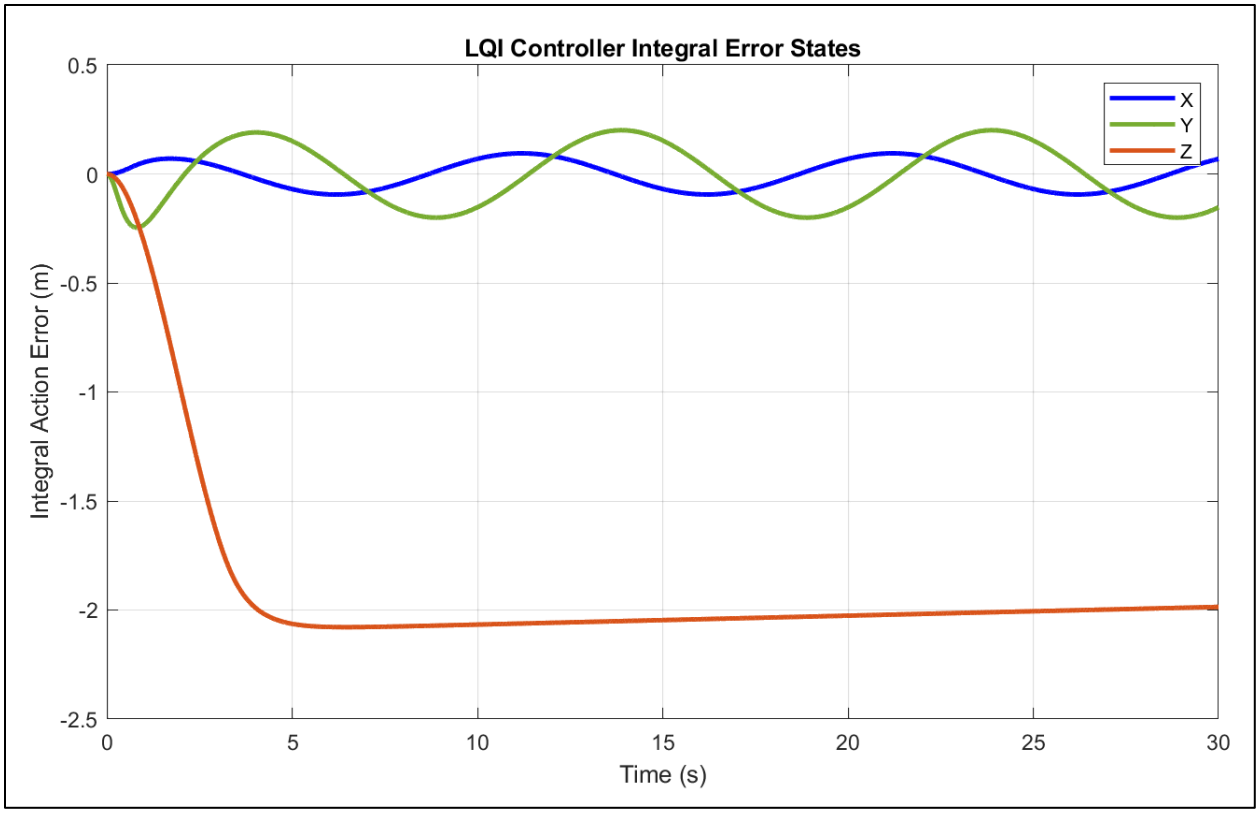

LQI Controller Integral Error States
Integral Action Error (m)
Time (s)
X
Y
Z
1
0
-1
-2
-3
-4
-5
0
1
2
3
4
5
6
7
8
9
10

Upwards Spiral Trajectory

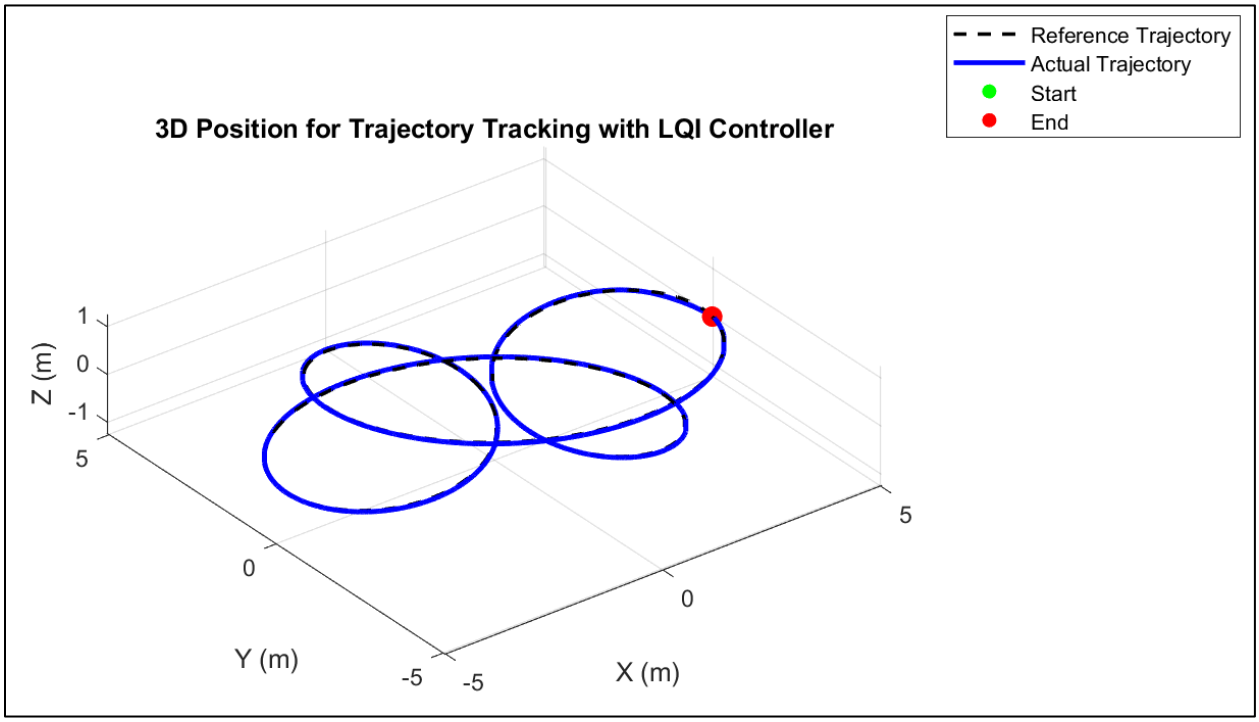


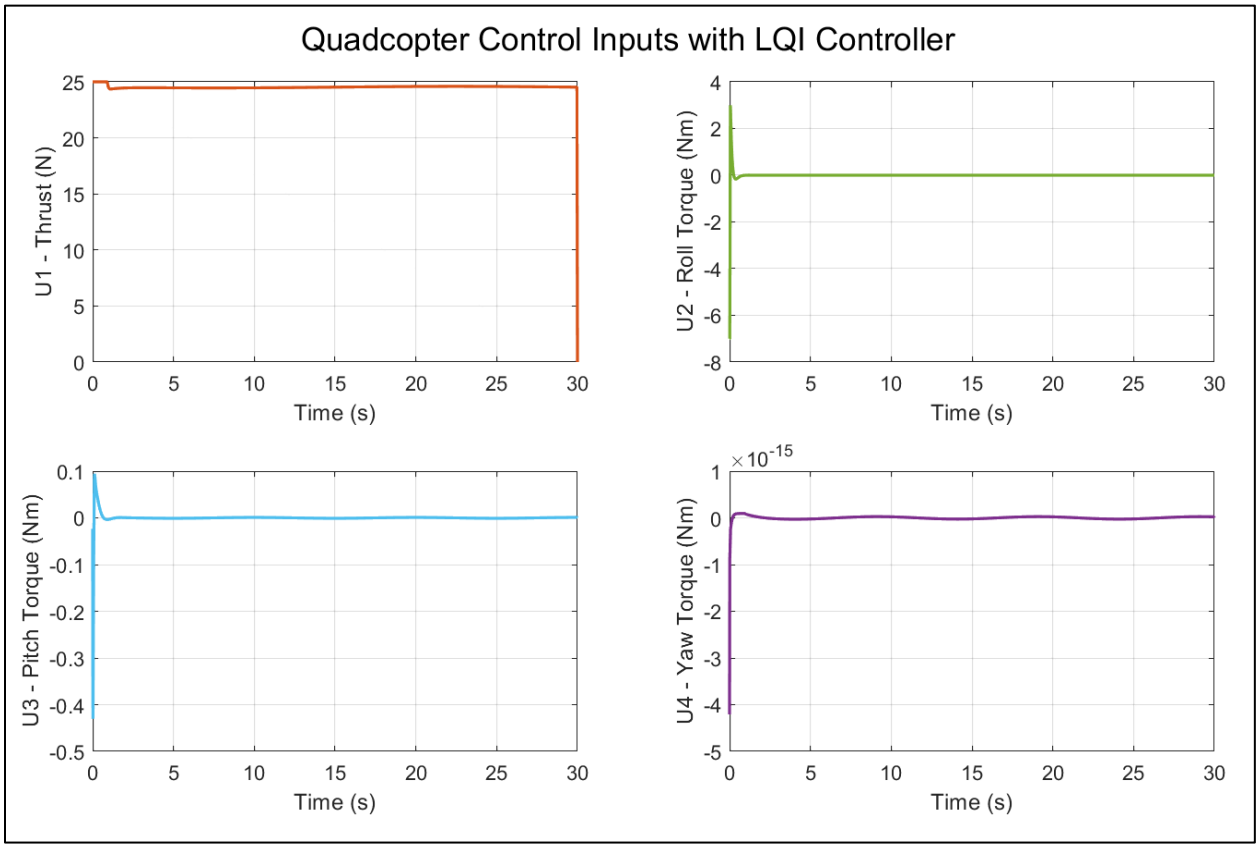

Quadcopter Control Inputs with LQI Controller
U1 - Thrust (N)
U2 - Roll Torque (Nm)
U3 - Pitch Torque (Nm)
U4 - Yaw Torque (Nm)
Time (s)

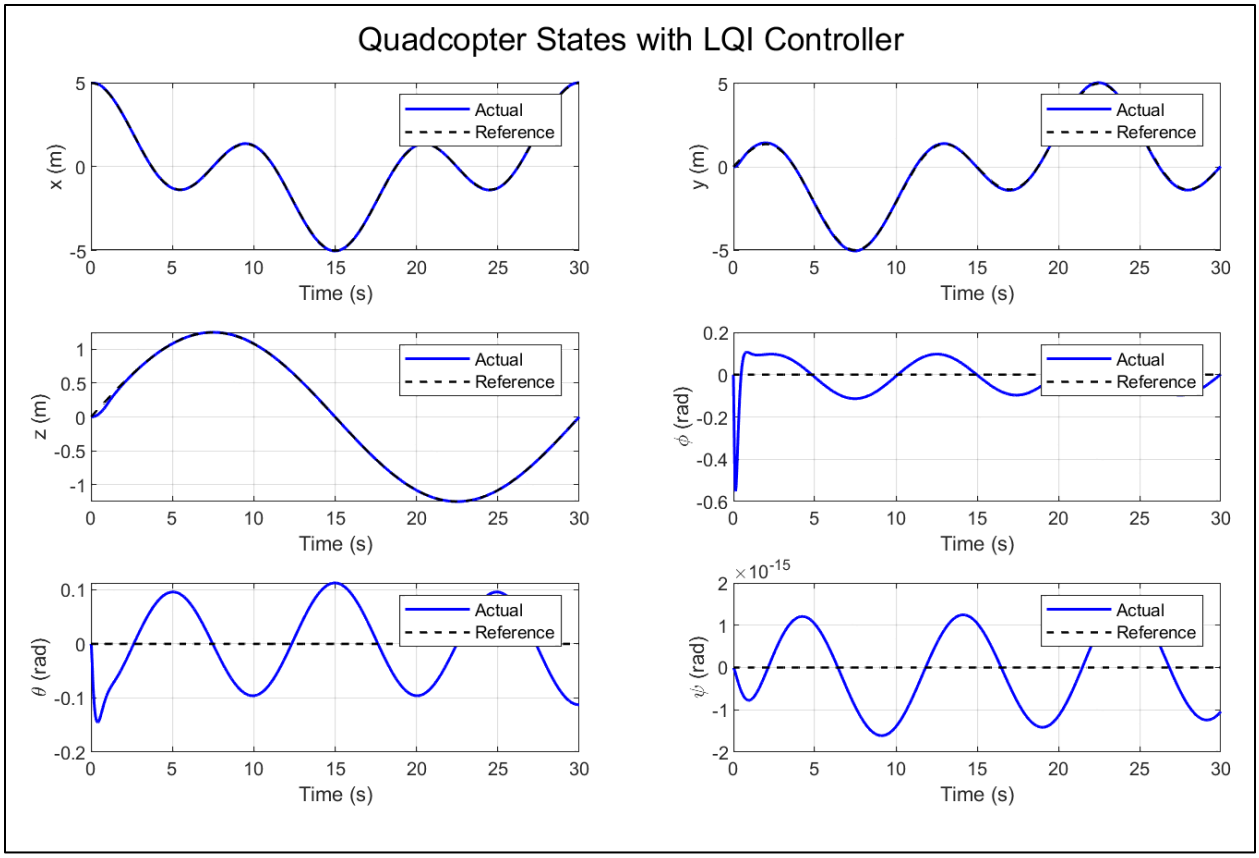

Quadcopter States with LQI Controller
x (m)
y (m)
z (m)
ϕ (rad)
θ (rad)
ψ (rad)
Actual
Reference
Time (s)

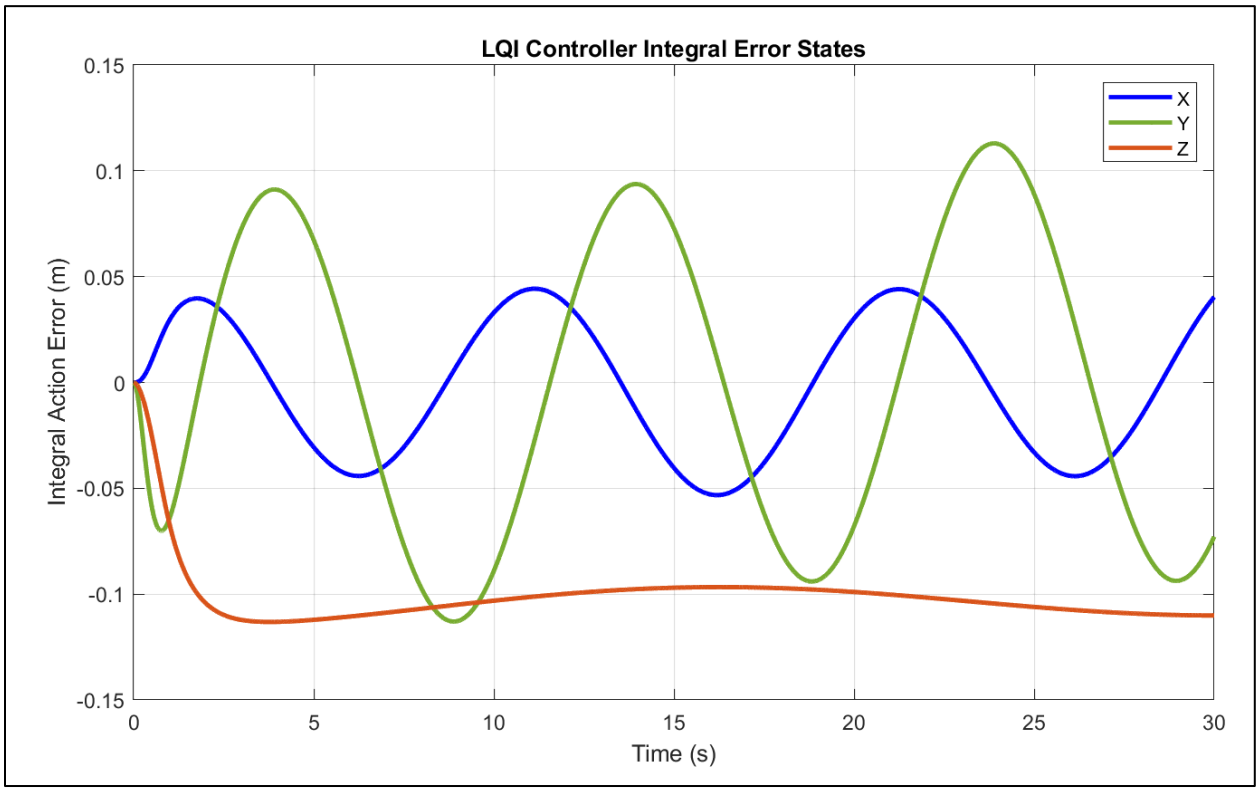

LQI Controller Integral Error States
Integral Action Error (m)
Time (s)
X
Y
Z
0.5
0
-0.5
-1
-1.5
-2
-2.5
0
5
10
15
20
25
30

Rose Petal Curve Trajectory

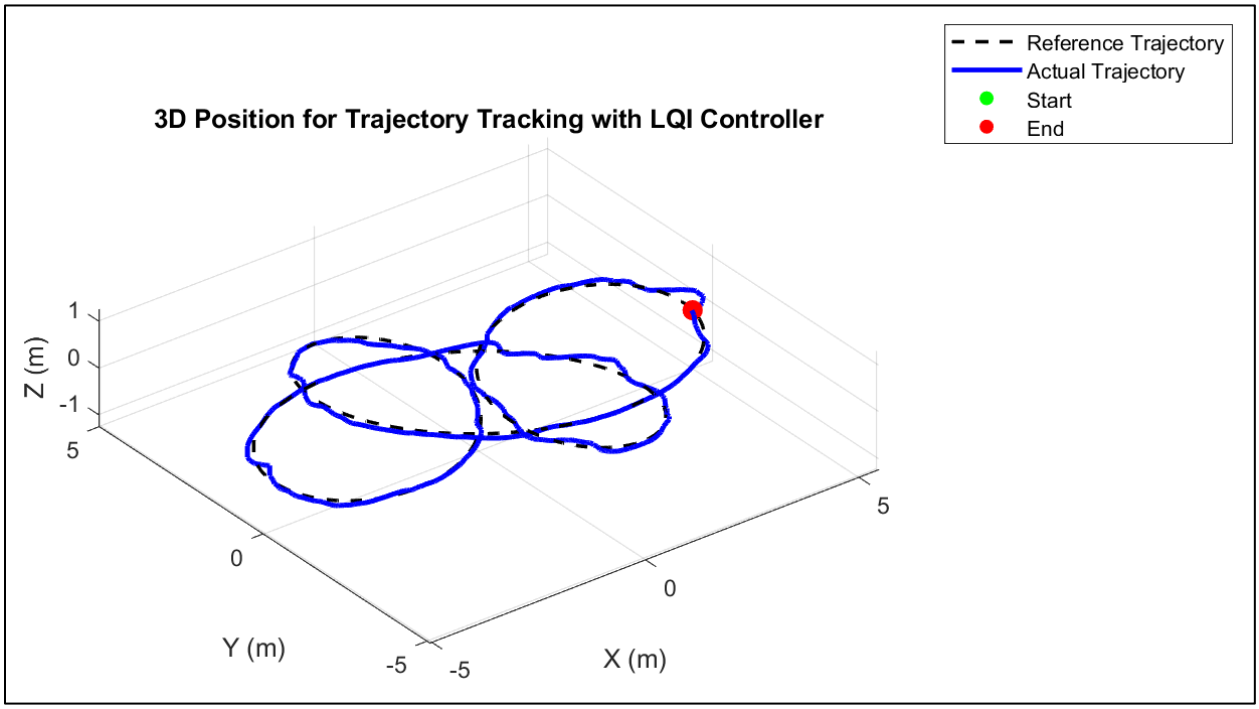


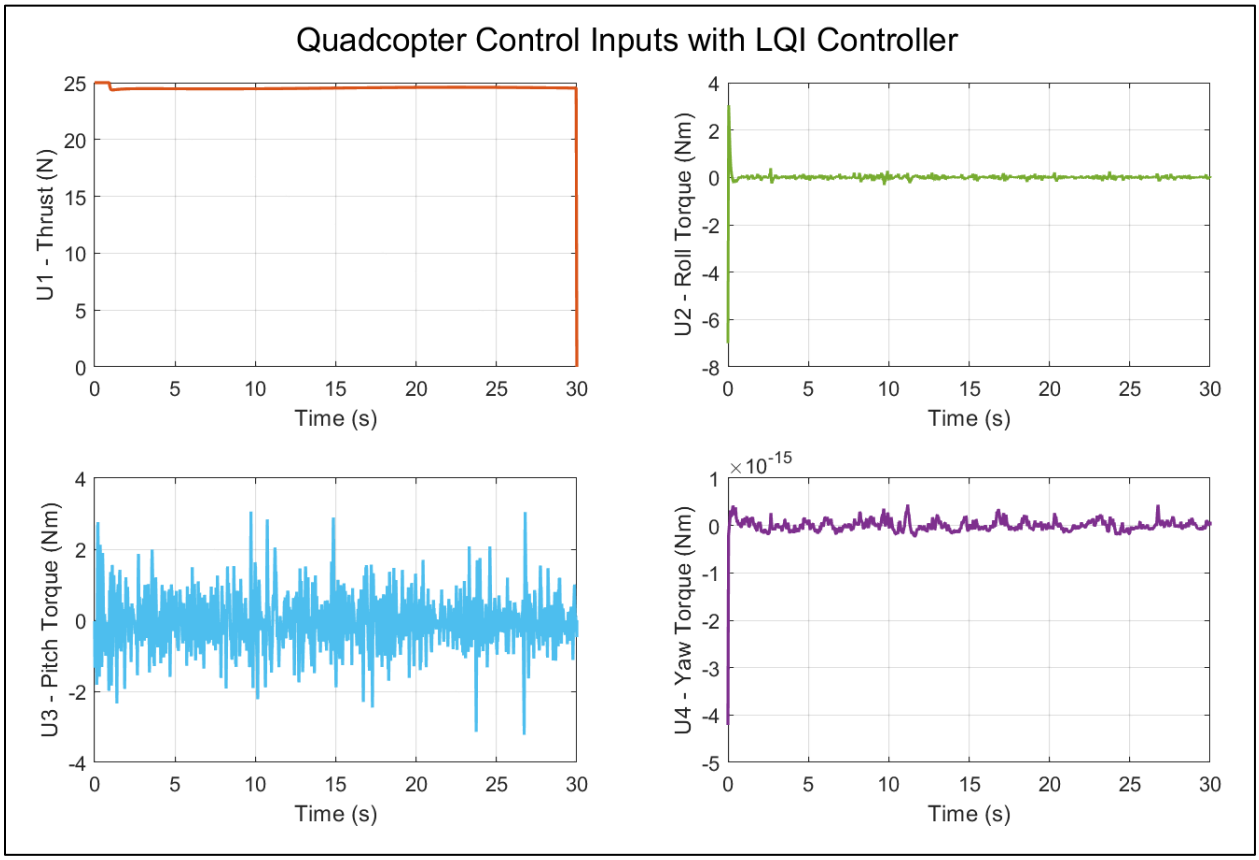


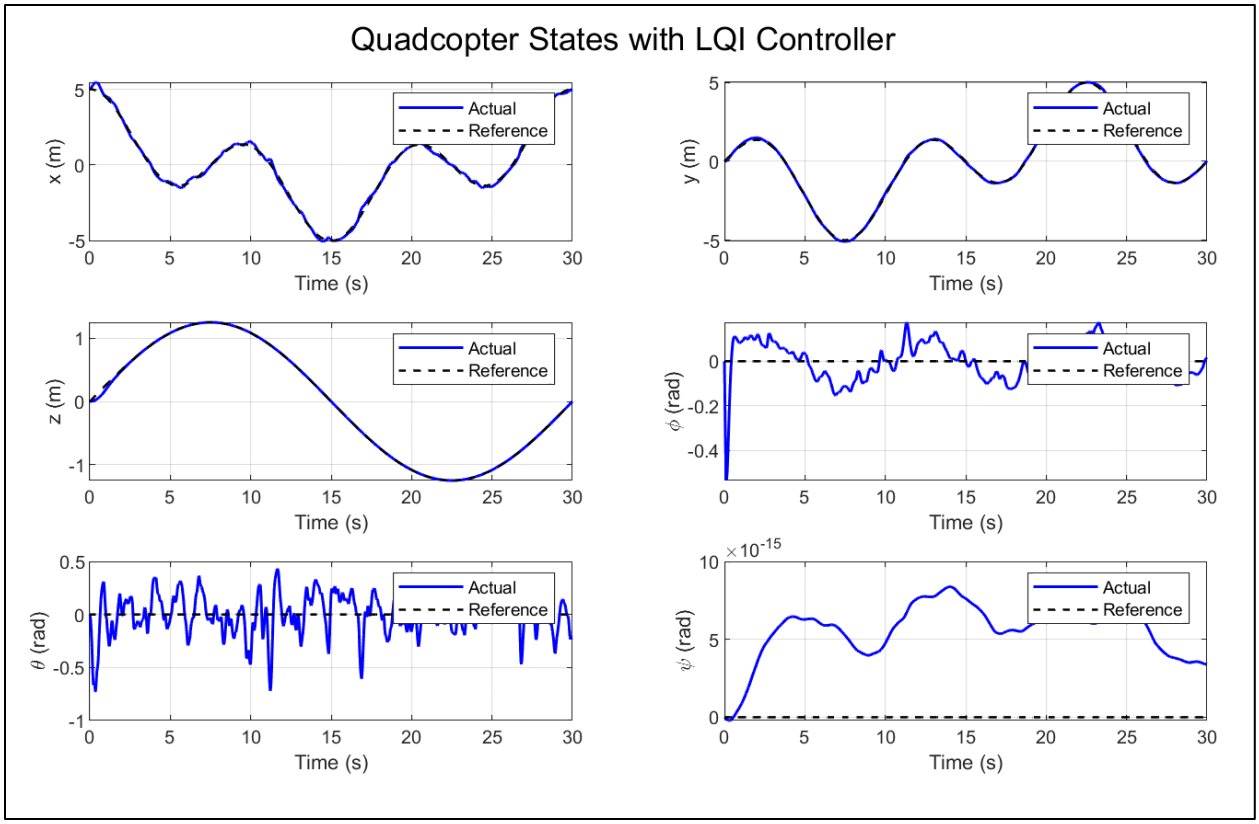

Quadcopter States with LQI Controller
x (m)
y (m)
z (m)
ϕ (rad)
θ (rad)
ψ (rad)
$\times 10^{-15}$
Actual
Reference
Time (s)

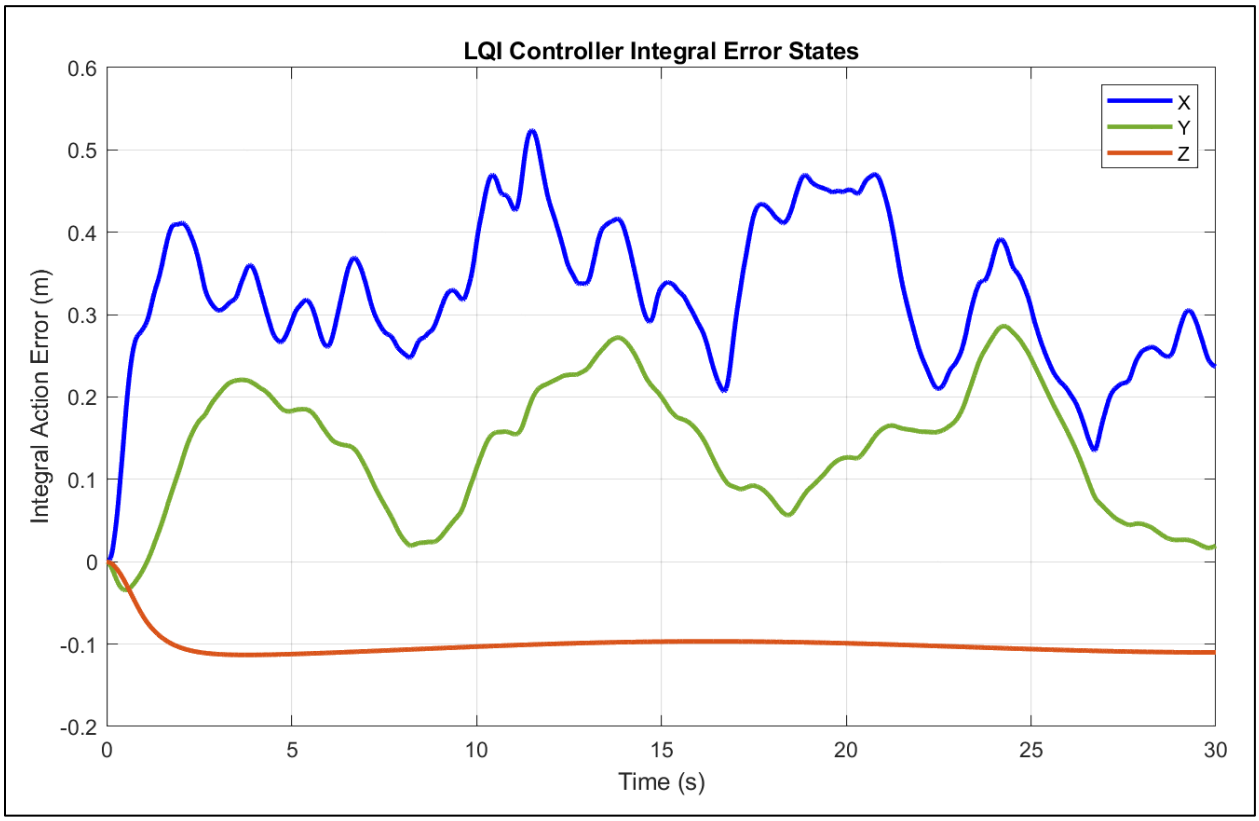

LQI Controller Integral Error States
Integral Action Error (m)
X
Y
Z
Time (s)

Rose Petal Curve Trajectory with Wind Disturbance



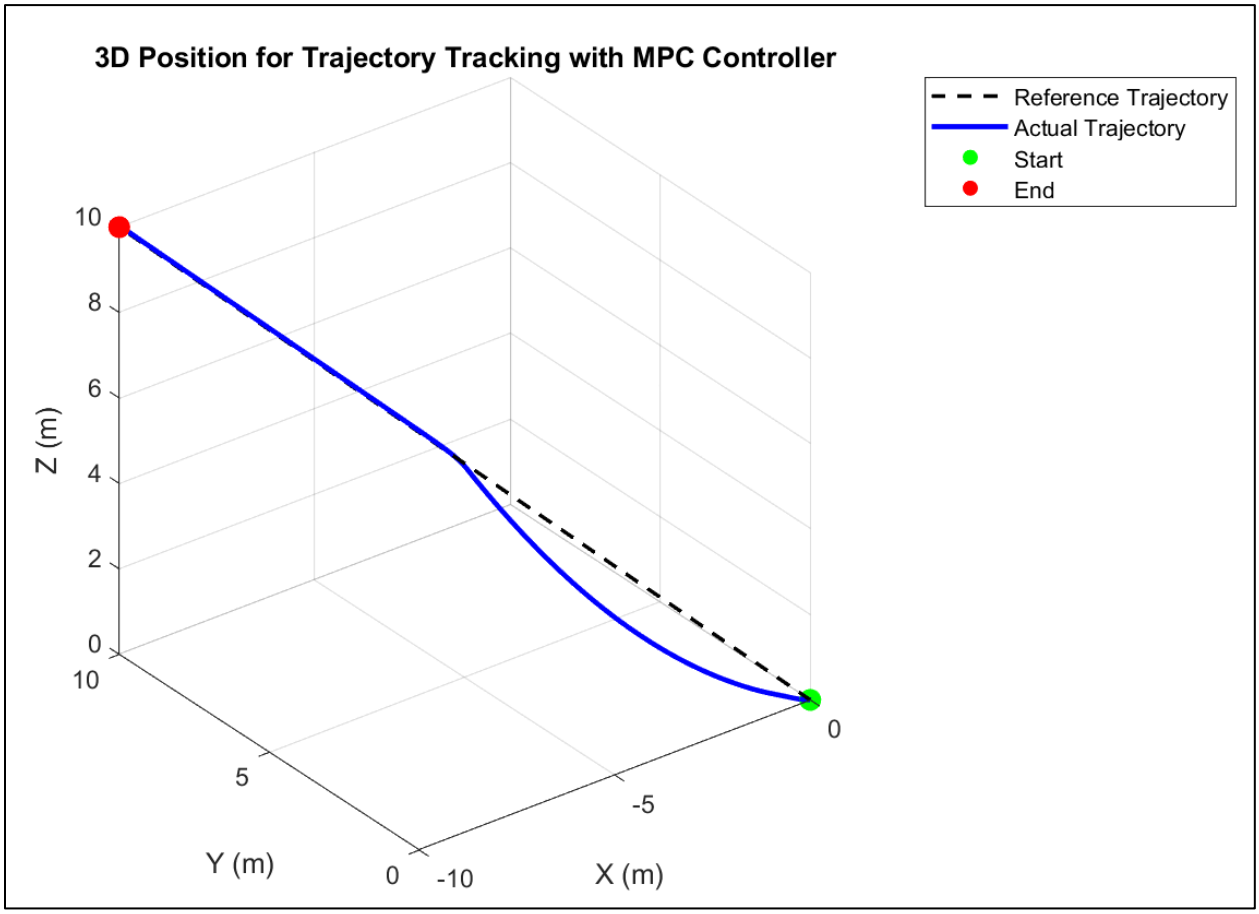


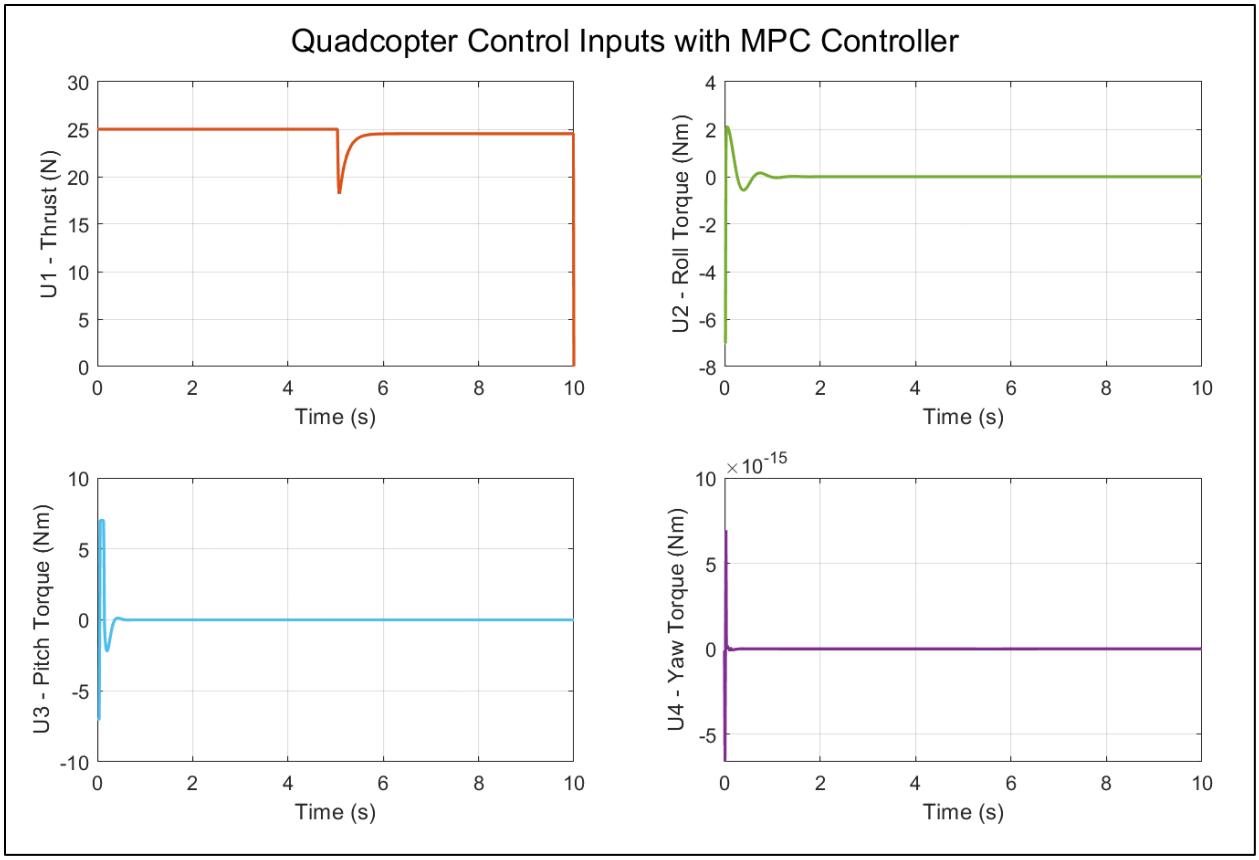

Quadcopter States with LQI Controller
x (m)
y (m)
z (m)
ϕ (rad)
θ (rad)
ψ (rad)
$\times 10^{-15}$
Actual
Reference
Time (s)

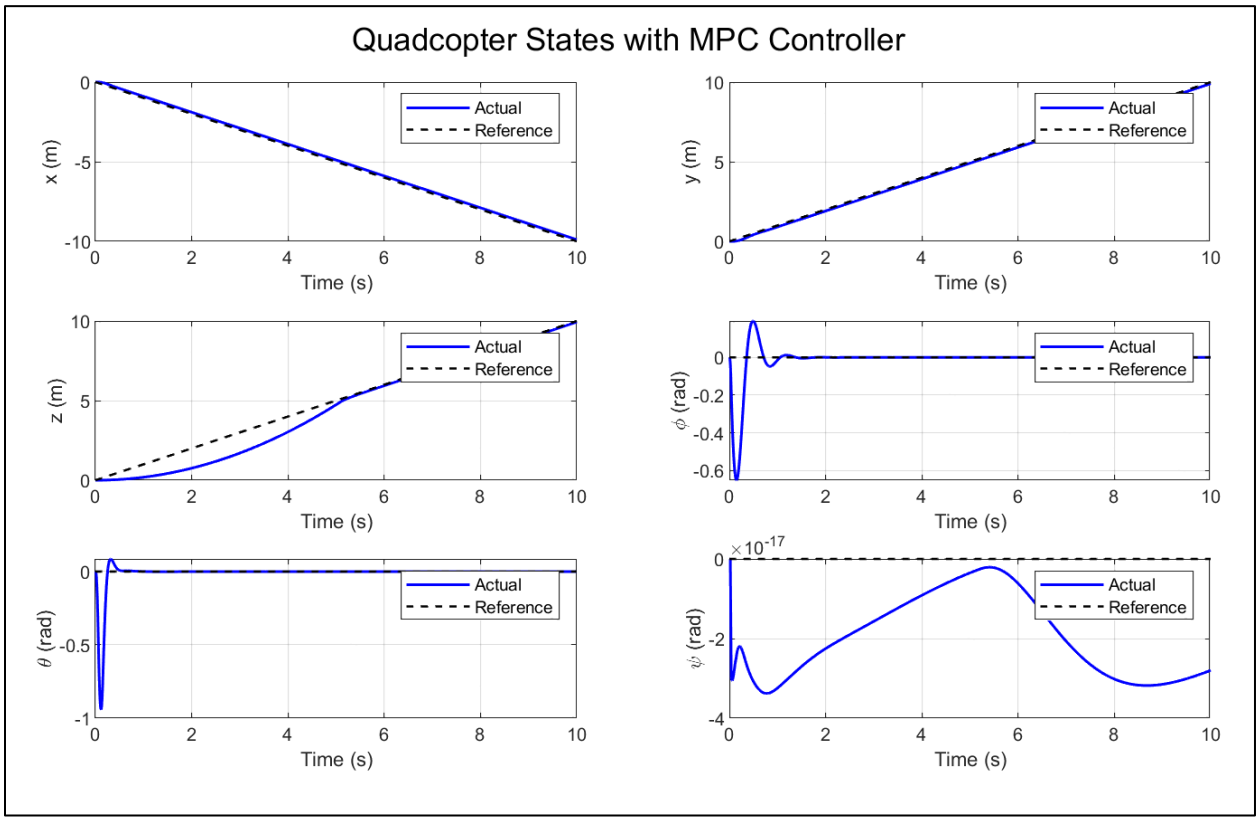

LQI Controller Integral Error States
Integral Action Error (m)
X
Y
Z
Time (s)

MPC Controller Performance Plots

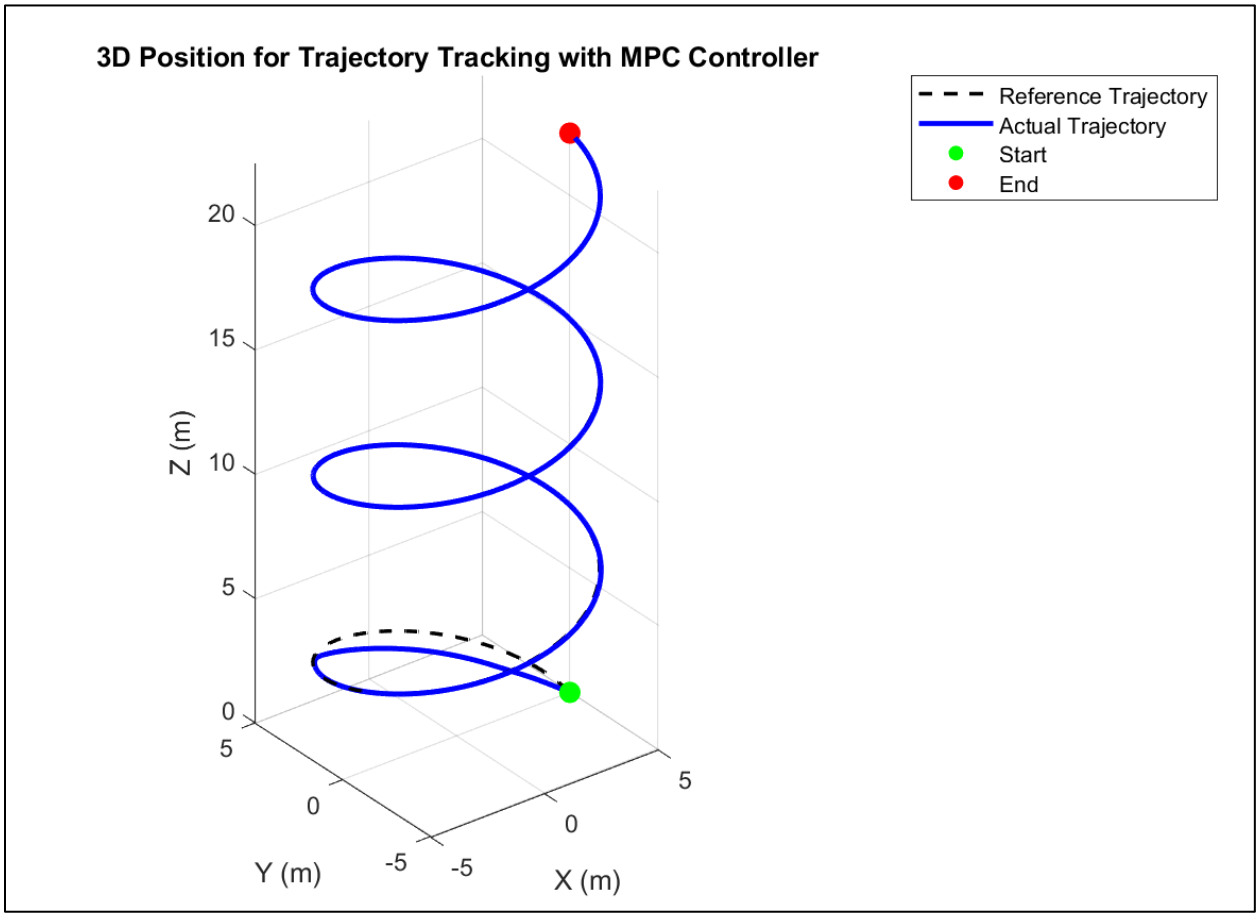
3D Line Trajectory

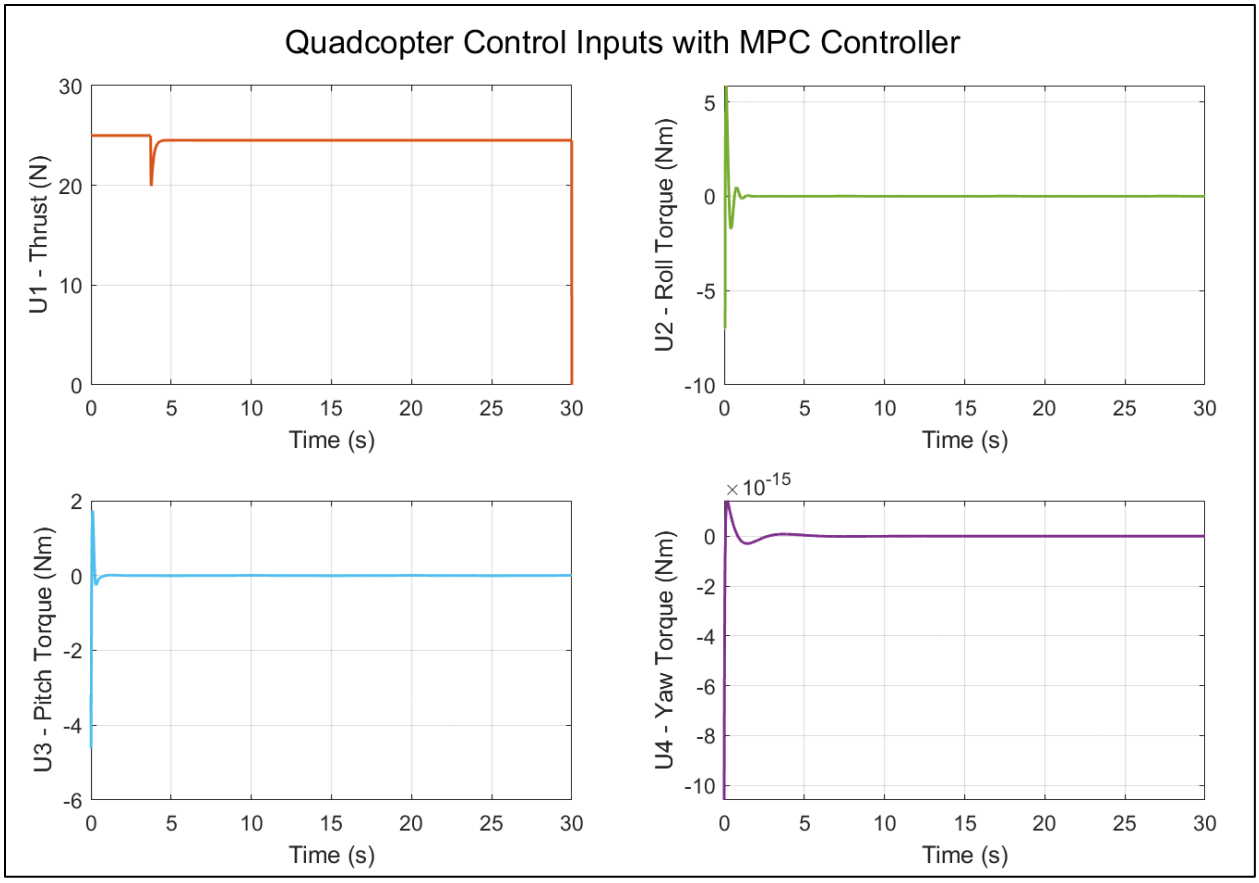


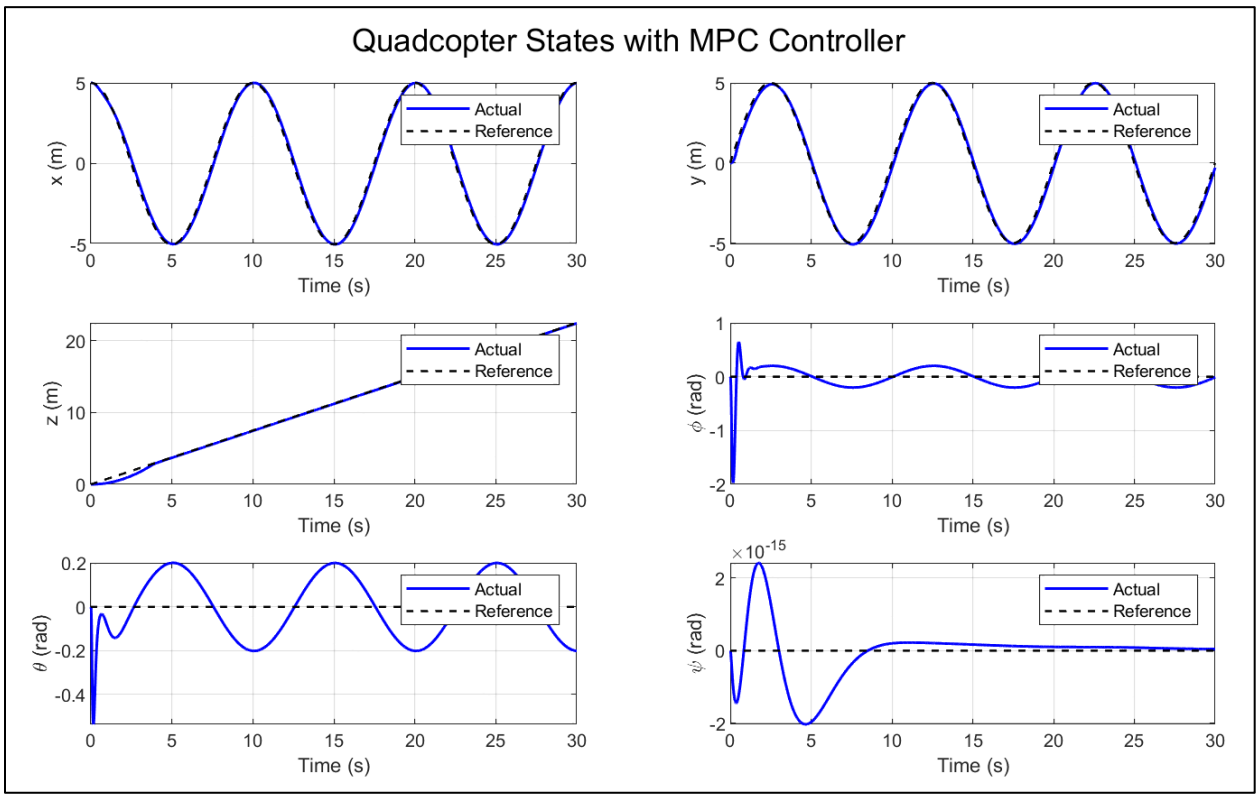

Quadcopter Control Inputs with MPC Controller
U1 - Thrust (N)
U2 - Roll Torque (Nm)
U3 - Pitch Torque (Nm)
U4 - Yaw Torque (Nm)
Time (s)
×10^-15


Quadcopter States with MPC Controller
x (m)
y (m)
z (m)
ϕ (rad)
θ (rad)
ψ (rad)
Actual
Reference
Time (s)
×10^-17

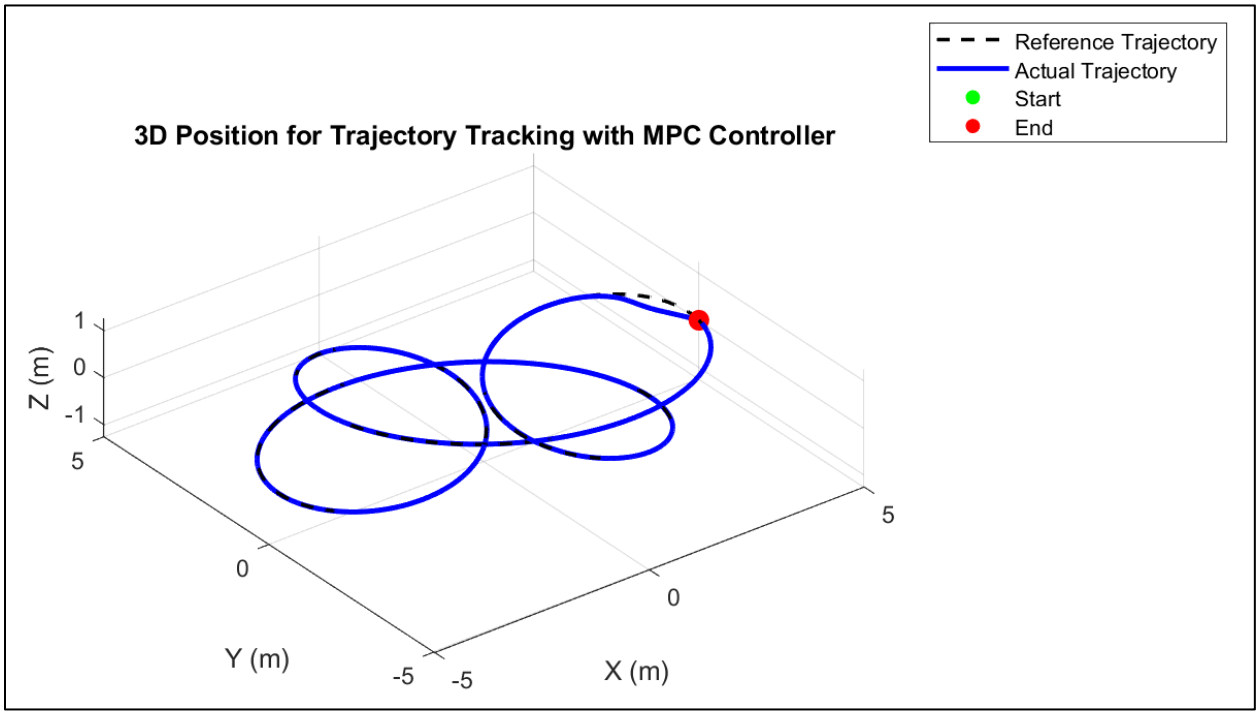
Upwards Spiral Trajectory

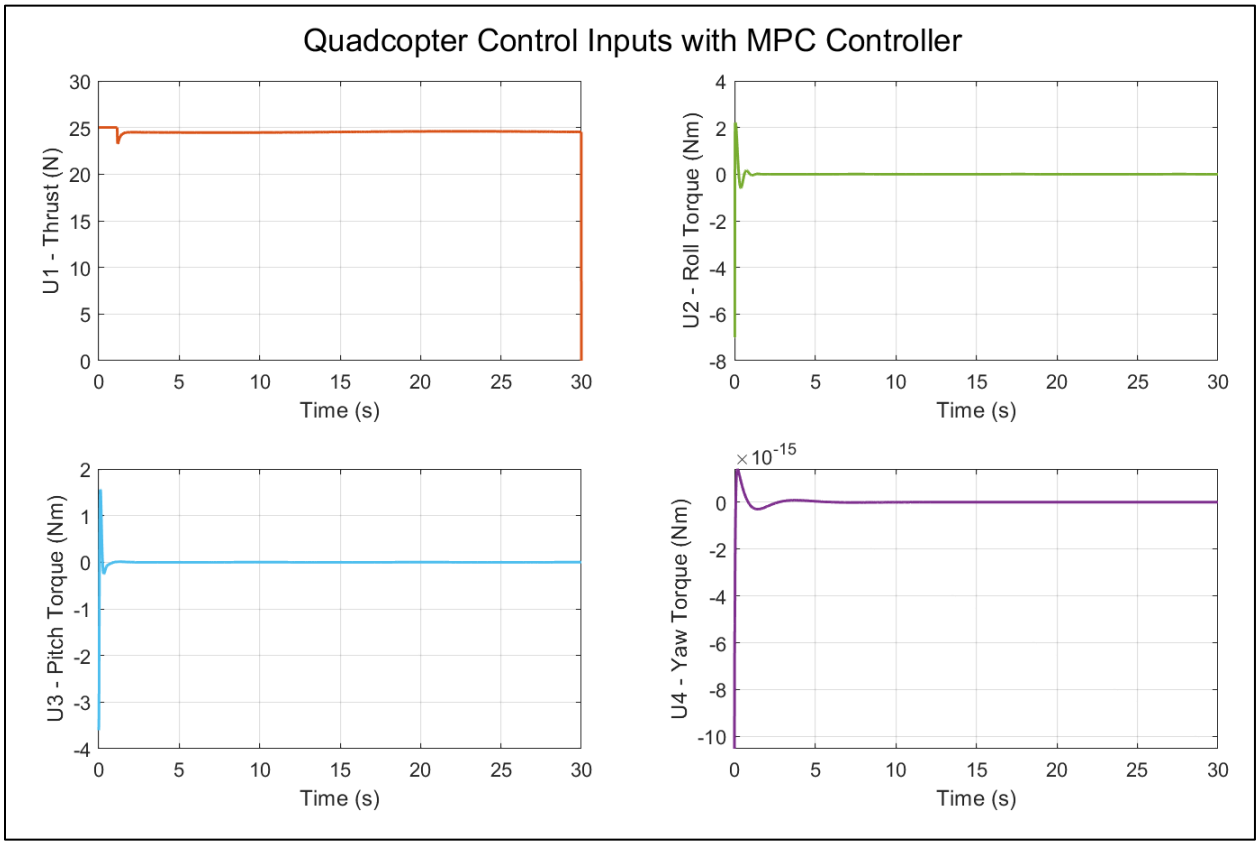


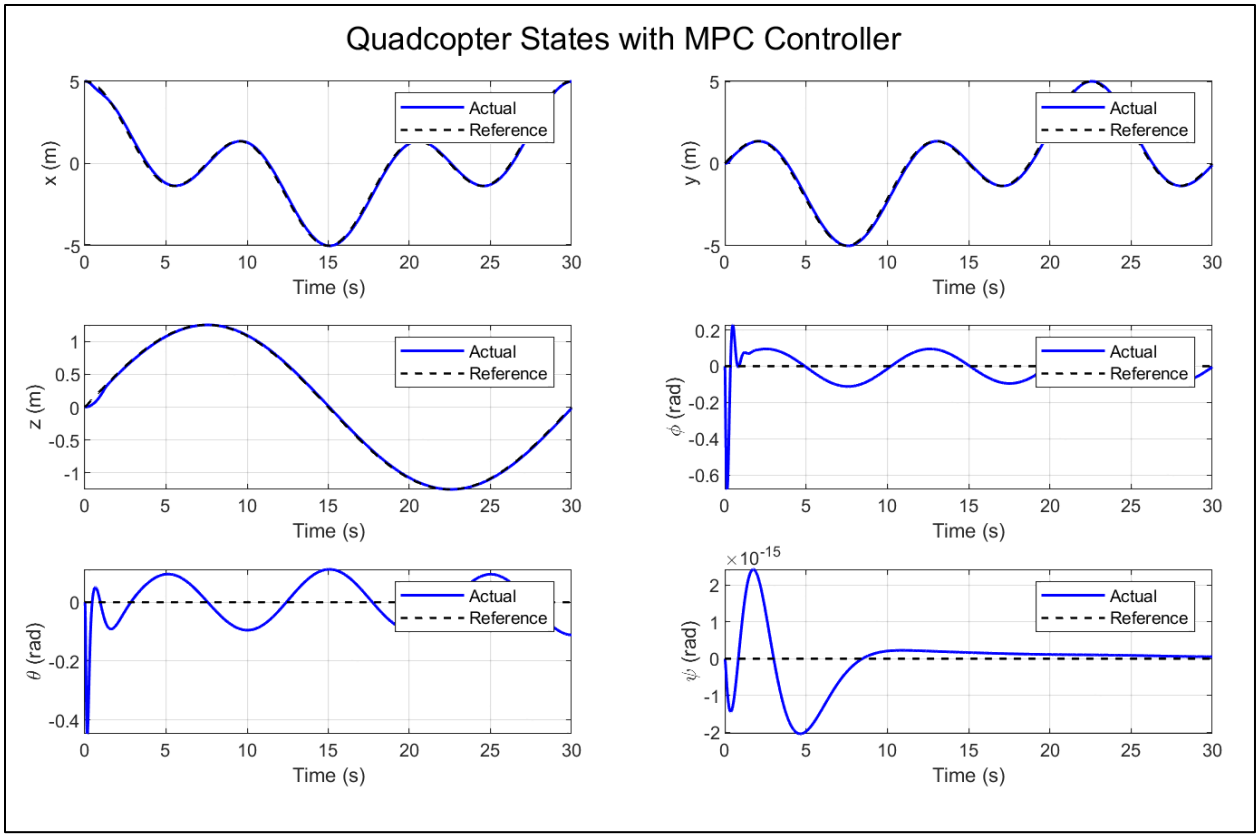

Quadcopter Control Inputs with MPC Controller
U1 - Thrust (N)
U2 - Roll Torque (Nm)
U3 - Pitch Torque (Nm)
U4 - Yaw Torque (Nm)
Time (s)
×10^-15


Quadcopter States with MPC Controller
x (m)
y (m)
z (m)
ϕ (rad)
θ (rad)
ψ (rad)
Actual
Reference
Time (s)
×10^-15

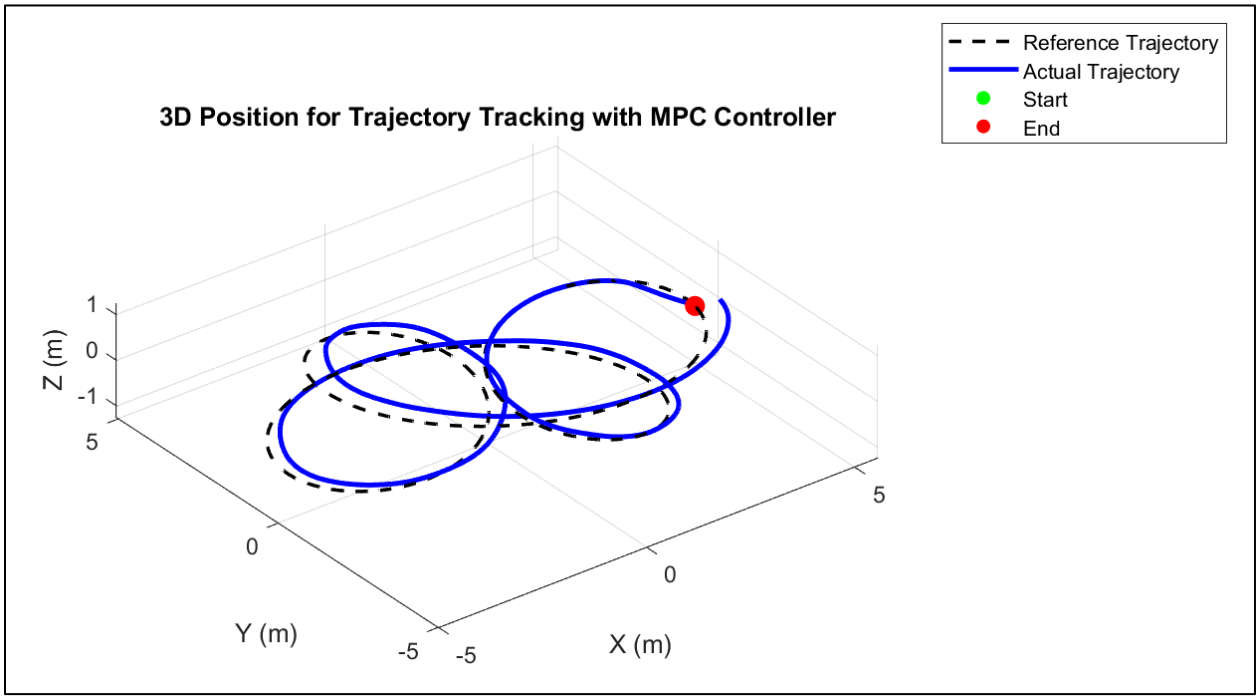
Rose Petal Curve Trajectory

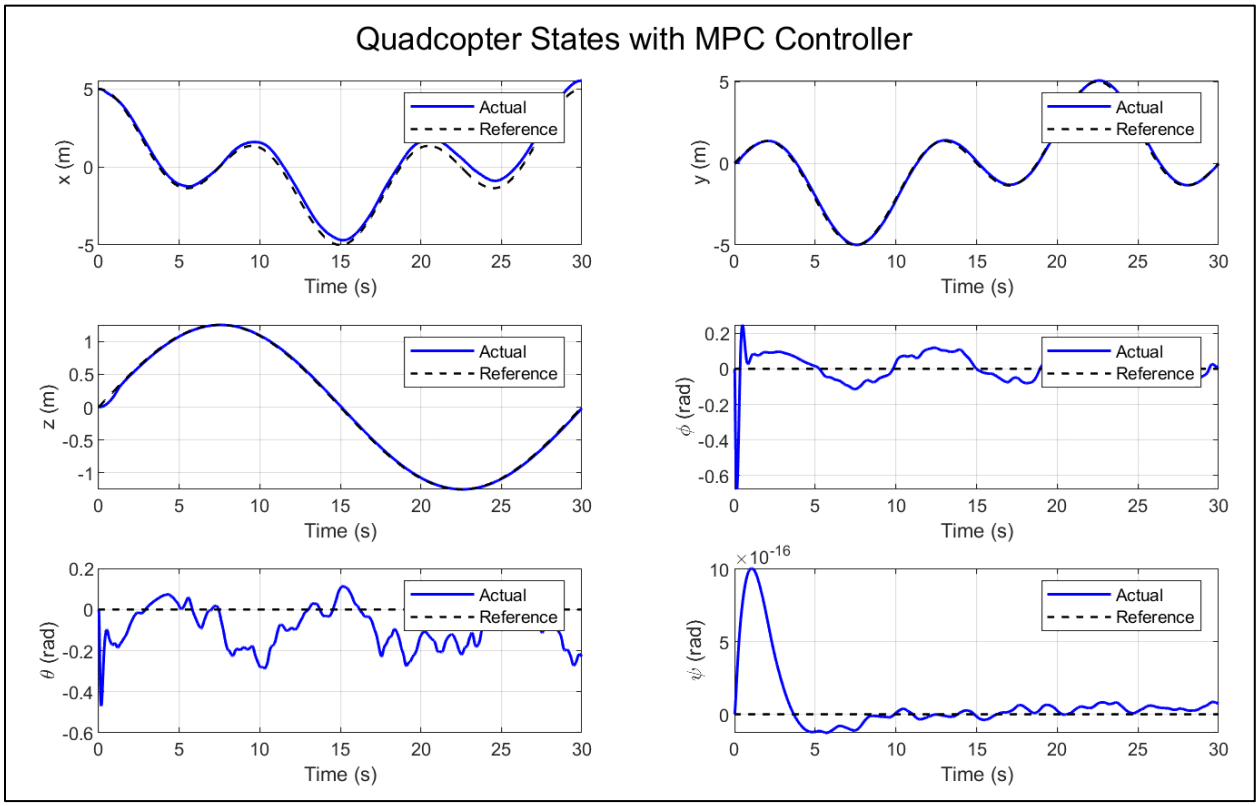


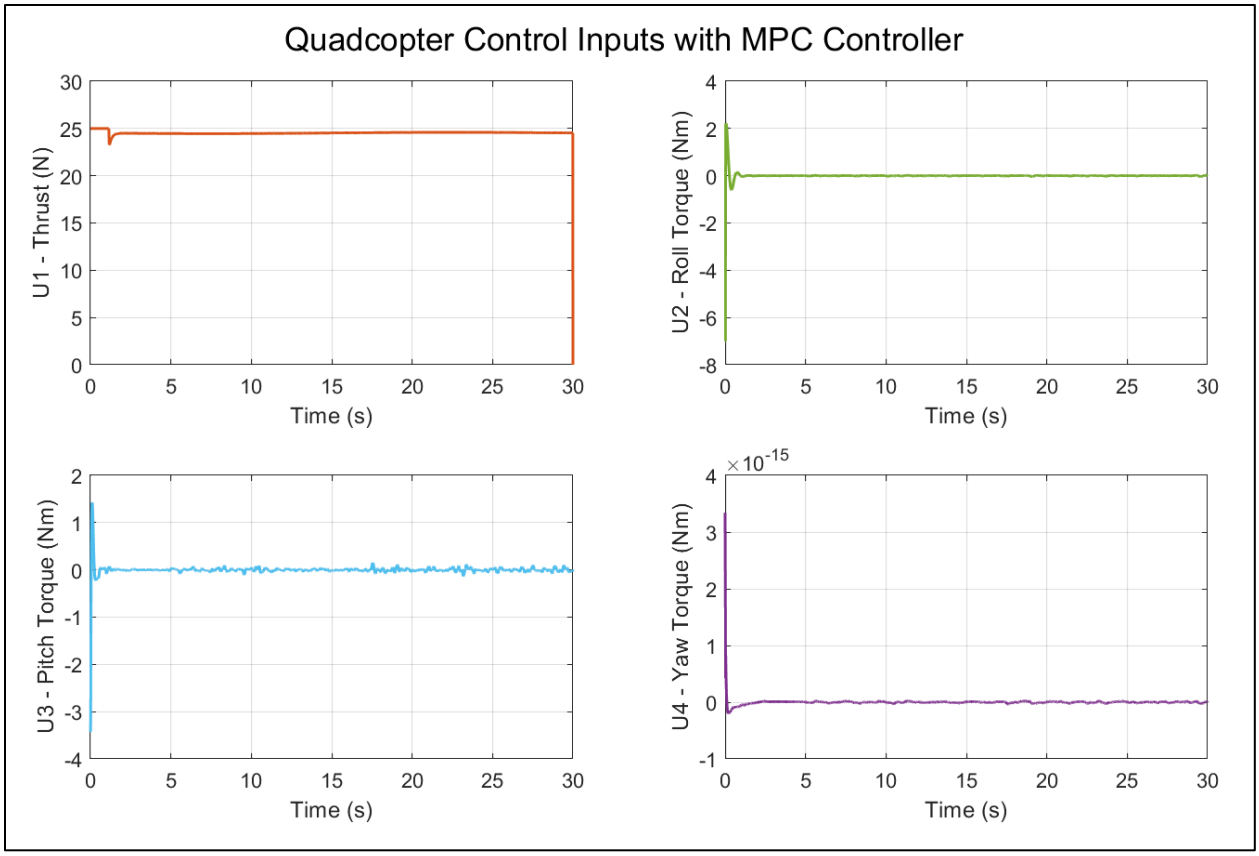



Quadcopter States with MPC Controller
x (m)
y (m)
z (m)
ϕ (rad)
θ (rad)
ψ (rad)
Time (s)
Actual
Reference
$\times 10^{-15}$

Rose Petal Curve Trajectory with Wind Disturbance






Quadcopter Control Inputs with MPC Controller
U1 - Thrust (N)
Time (s)
U2 - Roll Torque (Nm)
Time (s)
U3 - Pitch Torque (Nm)
Time (s)
U4 - Yaw Torque (Nm)
×10^-15
Time (s)