Université de Technologie de Compiègne

Département de Génie Informatique (GI)

ARS5 Project Report

"Quadcopter Model controlled using the Backstepping Controller"

Course: ARS5

Semester: Autumn 2025

Professor: Dr. Pedro CASTILLO GARCIA

Submitted by:
Stefano MUSSO PIANTELLI
Elias Charbel SALAMEH

Date: January 15, 2026

Contents

Acknowledgements

This project is a requirement for the ARS5 – Commande de robots autonomes en coopération. We would like to thank Dr. Pedro CASTILLO GARCIA for his remarkable efforts in delivering the material of this course and in helping us better improve in the control and automation fields.

Abstract

This project tackles the concepts of modeling and control of mobile robots such as the quadcopter. The control strategy is defined to be the backstepping non-linear control. The modeling technique is a free choice that we must make.

Keywords: Backstepping, Modeling, Stability Analysis, Quaternion, trajectory following

Chapter 1

Introduction

1.1 Problem Statement

The problem at hand is the modelling of a quadcopter system and to control it using backstepping control. This project is thus split into multiple tasks:

- Model the Quadcopter system
- Choose the control strategy
- Proceed with the thorough stability analysis
- Deploy the system in a controlled environment

1.2 Goals

The goals that must be accomplished at the end of the project are:

- the complete modelling of the Quadcopter to cover singularities and prepare the system to be controlled the best way possible.
- the stable and safe behavior of the quadcopter system
- the convergence to the constant targets
- the trajectory following based on a pre-defined 3D function

Chapter 2

Mathematical Non-Linear Model

2.1 Choice of the Model

The quadcopter system relies on non-linear dependencies in its physical equations. To model this non-linear system, one must choose one of the three known methods to represent the properties of the studied system:

- **Newton-Euler approach:** considers the flapping and the aerodynamics along with all forces that act on the system
- **Euler-Lagrange approach:** focusing on the potential and kinematic properties of the system without considering the perturbations
- **Quaternion approach:** solving singularities and allowing aggressive maneuvers to be applied

The method chosen is the **quaternion approach** since it allows for more applications based on its rich 4D (hyper-complex) representation and cove.

2.2 Quaternion Approach

2.2.1 Quaternion Translational Model

To define the translational model, we set the following state variable:

$$\mathbb{X}_{pos} = \begin{bmatrix} \xi^T & \dot{\xi}^T \end{bmatrix}^T$$

ξ being the inertial position vector and $\dot{\xi}$ being the inertial velocity vector.

Based on Newton's equations:

$$\mathbf{q} \otimes F_t \otimes \mathbf{q}^* = m\ddot{\xi} \tag{2.1}$$

such that F_t is the total force consisting of the control force or thrust $\hat{F}$ and the external forces F_{ext}. We can also add some drag.

The only force measured in the body frame is the thrust force u_z in the z-axis; thus, it needs conversion to the inertial coordinates using the appropriate rotation change. On the other hand, F_{ext} corresponds to the gravity force, for instance, and some drag forces along the longitudinal and lateral axis even if they are minimal.

Hence,

$$\dot{\mathbb{X}}_{pos} = \begin{bmatrix} \dot{\xi} \\ \ddot{\xi} \end{bmatrix} = \begin{bmatrix} \dot{\xi} \\ \mathbf{q} \otimes \frac{F_u}{m} \otimes \mathbf{q}^* + F_{ext} \end{bmatrix} \tag{2.2}$$

In equation 2.2,

$$F_{ext} = \begin{bmatrix} F_{drag,longitudinal} & F_{drag,lateral} & g \end{bmatrix}^T$$

2.2.2 Quaternion Rotational Model

To define the rotational model, we set the following state variables:

$$\mathbb{X}_{rot} = \begin{bmatrix} \mathbf{q}^T & \Omega^T \end{bmatrix}^T$$

where $\mathbf{q}$ is a quaternion and Ω is the rotational velocity matrix in body frame.

Based on quaternion algebra and specifically the derivative quaternion property applied to a rotation equation:

$$\dot{\mathbf{q}} = \frac{1}{2}\mathbf{q} \otimes \Omega \tag{2.3}$$

Based on Newton's equation, in particular on angular momentum for a rigid body:

$$\dot{\Omega} = I^{-1}(\tau - \Omega \times I\Omega) \tag{2.4}$$

where I is the inertial matrix:

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \tag{2.5}$$

Then, τ is the total torque moment, denoted by:

$$\tau = \sum_{i=1}^{4}(\tau_{M_i} + \tau_{r_i}) + \tau_d \tag{2.6}$$

Hence:

$$\dot{\mathbb{X}}_{rot} = \begin{bmatrix} \dot{\mathbf{q}} \\ \dot{\Omega} \end{bmatrix} = \begin{bmatrix} \frac{1}{2}\mathbf{q} \otimes \Omega \\ I^{-1}(\tau - \Omega \times I\Omega) \end{bmatrix} \tag{2.7}$$

in conclusion, the complete quadricopter model is the following:

$$\dot{\mathbb{X}} = \begin{bmatrix} \dot{\xi} \\ \ddot{\xi} \\ \dot{\mathbf{q}} \\ \dot{\Omega} \end{bmatrix} = \begin{bmatrix} \dot{\xi} \\ \mathbf{q} \otimes \frac{F_u}{m} \otimes \mathbf{q}^* + F_{ext} \\ \frac{1}{2}\mathbf{q} \otimes \Omega \\ I^{-1}(\tau - \Omega \times I\Omega) \end{bmatrix} \tag{2.8}$$

Chapter 3

Control Law

3.1 Choice

The control strategy is the backstepping control applied to the non-linear system to ensure that it converges. First, the highest degree state variables are converged to the desired position, which can be static or dynamic, than the remaining state variables are sequentially controlled until the control input is reached. Our system has six states (three positional states, three orientational states) and four inputs, the torques per axis and the equivalent thrust, since the axis of the rotors are colinear. This means that the studied system is under-actuated and we must use the state variables as intermediate virtual inputs to control the longitudinal and lateral position. This is actually clear since the quadcopter does not have any input that allows it to follow these two axis. However, by controlling the pitch, the longitudinal position will be impacted and by controlling the roll, the lateral position is acted on.

$$\ddot{\xi} \rightarrow \mathbf{e}_{\xi}(\xi, m) \rightarrow \dot{\xi}^{v} \rightarrow F_{u}(u, \mathbf{q}_{d}) \tag{3.1}$$

$$\mathbf{q} \rightarrow \mathbf{q}_{e}(\mathbf{q}, \mathbf{q}_{d}) \rightarrow \Omega^{v} \rightarrow \tau \tag{3.2}$$

Such that:

- $\mathbf{e}_{\xi}$ is the error between the robot and the setpoint (static or dynamic), can be extended to a multi-robot tracking error

- $\mathbf{q}_e$ is the quaternion error to converge $\mathbf{q}$ to $\mathbf{q}_d$ using the equation $\mathbf{q}_{\bar{e}} = \mathbf{q}_d^* \otimes \mathbf{q}$

It is a cascade structure separating the system into two subsystems: the translational dynamics (outer loop) and the rotational dynamics (inner loop).

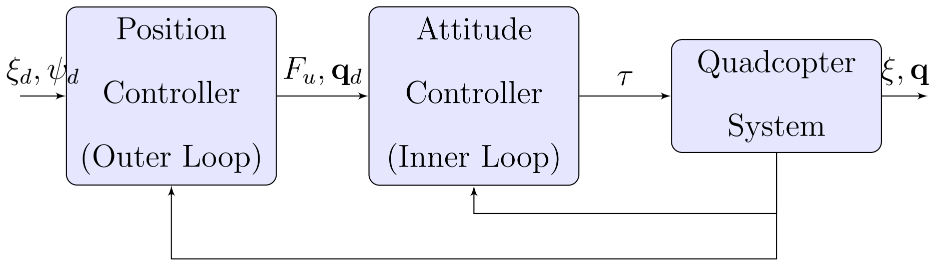

Figure 3.1: Cascade Control Architecture

The position controller generates the thrust F_u and the desired attitude $\mathbf{q}_d$ for the inner loop.

Backstepping Design The logic follows a recursive design procedure:

1. **Translational Control (Position):** The error is defined as $\mathbf{e}_\xi = \xi_d - \xi$. The virtual control input $\dot{\xi}^v$ is designed to stabilize the position error.

$$\xi, \dot{\xi} \xrightarrow{\text{Stabilization}} \mathbf{F}_{desired} \implies \begin{cases} F_u = ||\mathbf{F}_{desired}|| \\ \mathbf{q}_d = f(\mathbf{F}_{desired}) \end{cases} \tag{3.3}$$

2. **Rotational Control (Attitude):** The orientation error is defined using quaternion multiplication: $\mathbf{q}_e = \mathbf{q}_d^* \otimes \mathbf{q}$. The virtual angular velocity Ω^v is chosen to converge $\mathbf{q}$ to $\mathbf{q}_d$.

$$\mathbf{q}, \mathbf{q}_d \xrightarrow{\text{Error } \mathbf{q}_e} \Omega^v \xrightarrow{\text{Lyapunov}} \tau \tag{3.4}$$

Attitude Extraction The position controller generates a virtual control vector $\mathbf{U}_{pos} = [U_x, U_y, U_z]^T$ in the inertial frame representing the total desired force. From this vector, we must extract the control inputs for the inner loop: the thrust magnitude F_u and the desired attitude quaternion $\mathbf{q}_d$.

The thrust is obtained from the Euclidean norm of the virtual force:

$$F_u = ||\mathbf{U}_{pos}|| = \sqrt{U_x^2 + U_y^2 + U_z^2} \tag{3.5}$$

To determine $\mathbf{q}_d$, we construct a desired rotation matrix $\mathbf{R}_d = [\mathbf{x}_d, \mathbf{y}_d, \mathbf{z}_d]$. The desired body z-axis $\mathbf{z}_d$ must align with the thrust vector direction:

$$\mathbf{z}_d = \frac{\mathbf{U}_{pos}}{||\mathbf{U}_{pos}||} \tag{3.6}$$

Assuming a desired yaw angle ψ_d (usually 0 or provided by trajectory), we define an intermediate vector $\mathbf{l} = [-\sin(\psi_d), \cos(\psi_d), 0]^T$. The remaining axes are computed using cross products to ensure orthogonality:

$$\mathbf{x}_d = \frac{\mathbf{l} \times \mathbf{z}_d}{||\mathbf{l} \times \mathbf{z}_d||} \tag{3.7}$$

$$\mathbf{y}_d = \mathbf{z}_d \times \mathbf{x}_d \tag{3.8}$$

This formulation is a unique way to align the axes with the desired orientation.

Finally, the desired quaternion $\mathbf{q}_d$ is obtained by converting the rotation matrix $\mathbf{R}_d = [\mathbf{x}_d, \mathbf{y}_d, \mathbf{z}_d]$ into quaternion form.

3.2 Stability Analysis

3.2.1 Translational Part

To apply the concepts listed in section 3.1:

$$\dot{\bar{\xi}} = \bar{v} \tag{3.9}$$

$$\ddot{\bar{\xi}} = \dot{\bar{v}} = \mathbf{q} \otimes \frac{F_u}{m} \otimes \mathbf{q}^* + F_{ext} \tag{3.10}$$

The tracking error between the robot and the target is:

$$\bar{\mathbf{e}}_{\xi,1} = -[\mathbf{a} * (\bar{\xi} - \bar{\xi}_{target}) - \mathbf{d}] \tag{3.11}$$

This equation maintains a distance parameter based on the radius of the drone. It also allows to add more robots to follow in future steps.

With $\mathbf{a}, \mathbf{d} \geq 0$ Thus,

$$\dot{\bar{\mathbf{e}}}_{\xi,1} = -\mathbf{a} * (\dot{\bar{\xi}} - \dot{\bar{\xi}}_{target}) \tag{3.12}$$

Proposing a positive function:

$$V_1(\bar{\mathbf{e}}_{\xi,1}) = \frac{\bar{\mathbf{e}}_{\xi,1}^T \bar{\mathbf{e}}_{\xi,1}}{2} \tag{3.13}$$

Which yields:

$$\dot{V}_1(\bar{\mathbf{e}}_{\xi,1}) = \bar{\mathbf{e}}_{\xi,1}^T \dot{\bar{\mathbf{e}}}_{\xi,1} = \bar{\mathbf{e}}_{\xi,1}^T * [-\mathbf{a}(\dot{\bar{\xi}} - \dot{\bar{\xi}}_{target})] \tag{3.14}$$

We propose:

$$\dot{\bar{\xi}}^v = \frac{k_1 \bar{\mathbf{e}}_{\xi,1}}{a} + \dot{\bar{\xi}}_{target} \tag{3.15}$$

by ensuring that $\dot{\bar{\xi}} \to \dot{\bar{\xi}}^v$. This means that $\dot{V}_1(\bar{\mathbf{e}}_{\xi,1}) \leq 0$. Let's write the new variable $\bar{\mathbf{e}}_{\xi,2} = \dot{\bar{\xi}} - \dot{\bar{\xi}}^v$.

We write the new system:

$$\begin{cases} \dot{\bar{\mathbf{e}}}_{\xi,1} = -\mathbf{a}.(\dot{\bar{\xi}}^v + \bar{\mathbf{e}}_{\xi,2} - \dot{\bar{\xi}}_{target}) = -\mathbf{a}.\bar{\mathbf{e}}_{\xi,2} - k_1 \bar{\mathbf{e}}_{\xi,1} \\ \dot{\bar{\mathbf{e}}}_{\xi,2} = \dot{\bar{\xi}} - \ddot{\bar{\xi}}^v = \mathbf{q} \otimes \frac{F_u}{m} \otimes \mathbf{q}^* + F_{ext} - (\frac{k_1 \dot{\bar{\mathbf{e}}}_{\xi,1}}{a} + \ddot{\bar{\xi}}_{target}) \end{cases} \tag{3.16}$$

Thus,

$$\begin{cases} \dot{\bar{\mathbf{e}}}_{\xi,1} = -\mathbf{a}.\bar{\mathbf{e}}_{\xi,2} - k_1 \bar{\mathbf{e}}_{\xi,1} \\ \dot{\bar{\mathbf{e}}}_{\xi,2} = \mathbf{q} \otimes \frac{F_u}{m} \otimes \mathbf{q}^* + F_{ext} - (\frac{k_1 \dot{\bar{\mathbf{e}}}_{\xi,1}}{a} + \ddot{\bar{\xi}}_{target}) \end{cases} \tag{3.17}$$

We propose a positive function:

$$V(\bar{\mathbf{e}}_{\xi,1}, \bar{\mathbf{e}}_{\xi,2}) = \frac{\bar{\mathbf{e}}_{\xi,1}^T \bar{\mathbf{e}}_{\xi,1}}{2} + \frac{\bar{\mathbf{e}}_{\xi,2}^T \bar{\mathbf{e}}_{\xi,2}}{2} \tag{3.18}$$

Which yields:

$$\dot{V}(\bar{\mathbf{e}}_{\xi,1}, \bar{\mathbf{e}}_{\xi,2}) = \bar{\mathbf{e}}_{\xi,1}^T \dot{\bar{\mathbf{e}}}_{\xi,1} + \bar{\mathbf{e}}_{\xi,2}^T \dot{\bar{\mathbf{e}}}_{\xi,2} \tag{3.19}$$

$$\dot{V}(\bar{\mathbf{e}}_{\xi,1}, \bar{\mathbf{e}}_{\xi,2}) = \bar{\mathbf{e}}_{\xi,1}^T.[-\mathbf{a}.\bar{\mathbf{e}}_{\xi,2} - k_1\bar{\mathbf{e}}_{\xi,1}] + \bar{\mathbf{e}}_{\xi,2}^T.[\mathbf{q} \otimes \frac{F_u}{m} \otimes \mathbf{q}^* + F_{ext} - (\frac{k_1\dot{\bar{\mathbf{e}}}_{\xi,1}}{a} + \ddot{\bar{\xi}}_{target})] \tag{3.20}$$

We simplify:

$$\dot{V}(\bar{\mathbf{e}}_{\xi,1}, \bar{\mathbf{e}}_{\xi,2}) = -k_1\bar{\mathbf{e}}_{\xi,1}^T\bar{\mathbf{e}}_{\xi,1} + \bar{\mathbf{e}}_{\xi,2}^T.[-\mathbf{a}.\bar{\mathbf{e}}_{\xi,1} + \mathbf{q} \otimes \frac{F_u}{m} \otimes \mathbf{q}^* + F_{ext} - (\frac{k_1\dot{\bar{\mathbf{e}}}_{\xi,1}}{a} + \ddot{\bar{\xi}}_{target})] \tag{3.21}$$

Hence,

$$\mathbf{q} \otimes \frac{F_u}{m} \otimes \mathbf{q}^* = \mathbf{a}.\bar{\mathbf{e}}_{\xi,1} - F_{ext} + (\frac{k_1\dot{\bar{\mathbf{e}}}_{\xi,1}}{a} + \ddot{\bar{\xi}}_{target}) - k_2\bar{\mathbf{e}}_{\xi,2} \tag{3.22}$$

Therefore,

$$F_u = \mathbf{q}^{-1} \otimes [m.(\mathbf{a}.\bar{\mathbf{e}}_{\xi,1} - F_{ext} + (\frac{k_1\dot{\bar{\mathbf{e}}}_{\xi,1}}{a} + \ddot{\bar{\xi}}_{target}) - k_2\bar{\mathbf{e}}_{\xi,2})] \otimes {\mathbf{q}^*}^{-1} \tag{3.23}$$

This control input ensures that

$$\dot{V}(\bar{\mathbf{e}}_{\xi,1}, \bar{\mathbf{e}}_{\xi,2}) = -k_1\bar{\mathbf{e}}_{\xi,1}^T\bar{\mathbf{e}}_{\xi,1} - k_2\bar{\mathbf{e}}_{\xi,2}^T\bar{\mathbf{e}}_{\xi,2} \leq 0 \tag{3.24}$$

Stability is ensured using backstepping control. This way, by setting the control input as seen above, the system will converge the velocity to its virtual input role for the position which converges when the positional error $\bar{\mathbf{e}}_{\xi,1}$ is minimized.

3.2.2 Rotational part

First of all, we make a change of variable introducing the quaternion error and its derivate:

$$\mathbf{q}_{\bar{e}} = \mathbf{q}_d^*\mathbf{q} \tag{3.25}$$

$$\dot{\mathbf{q}_\mathbf{e}} = \frac{1}{2}M(\mathbf{q}_{\bar{e}})\Omega \tag{3.26}$$

Proposing the following positive function:

$$V_1(\mathbf{q}_{\bar{e}}) = \frac{1}{2}\mathbf{q}_e^T\mathbf{q}_e \tag{3.27}$$

And its derivative is:

$$\dot{V}_1(\mathbf{q}_e) = \mathbf{q}_e^T\dot{\mathbf{q}}_e = \frac{1}{2}\mathbf{q}_e^T M(\mathbf{q}_e)\Omega \tag{3.28}$$

The virtual input is:

$$\Omega \to (\Omega^v = -2k_3 M^T\mathbf{q}_e) \tag{3.29}$$

with $k_3 \geq 0$. This means that:

$$\dot{V}_1(\mathbf{q}_e) = -\mathbf{q}_e^T M(\mathbf{q}_{\bar{e}})k_3 M^T(\mathbf{q}_{\bar{e}})\mathbf{q}_e \leq 0 \tag{3.30}$$

Proposing the following error and its derivative:

$$\alpha_2 = \Omega - \Omega^v \tag{3.31}$$

$$\dot{\alpha_2} = \dot{\Omega} - \dot{\Omega^v} \tag{3.32}$$

Such that:

$$\begin{aligned}
\dot{\Omega^v} &= \frac{d}{dt}(-2k_3 M^T\mathbf{q}_e) \\
\dot{\Omega^v} &= -2k_3(\dot{M^T}\mathbf{q}_e + M^T\dot{\mathbf{q}}_e) \\
\dot{\Omega^v} &= -2k_3(M^T(\dot{\mathbf{q}e})\mathbf{q}_e + M^T\dot{\mathbf{q}}_e)
\end{aligned}$$

It follows that:

$$\dot{\mathbf{q}}_e = \frac{1}{2}M(\mathbf{q}_{\bar{e}})(\Omega^v + \alpha_2) \tag{3.33}$$

$$I\dot{\alpha}_2 = -\Omega \times I\Omega + \tau - I\dot{\Omega}^v \tag{3.34}$$

Proposing the following positive function:

$$V(\mathbf{q}_e, \alpha_2) = \frac{1}{2}\mathbf{q}_e^T\mathbf{q}_e + \frac{1}{2}\alpha_2^T I \alpha_2 \tag{3.35}$$

then

$$\dot{V}(\mathbf{q}_e, \alpha_2) = \frac{1}{2}\mathbf{q}_e^T M(\mathbf{q}_{\bar{e}})(\Omega^v + \alpha_2) + \alpha_2^T(-\Omega \times I\Omega + \tau - I\dot{\Omega}^v) \tag{3.36}$$

Propose the following control:

$$\tau = \Omega \times I\Omega + I\dot{\Omega}^v - k_4\alpha_2 - \frac{1}{2}M(\mathbf{q}_{\bar{e}})\mathbf{q}_e^T \tag{3.37}$$

Therefore:

$$\dot{V}(\mathbf{q}_e, \alpha_2) = -\mathbf{q}_e M(\mathbf{q}_{\bar{e}}) k_3 M(\mathbf{q}_{\bar{e}})^T \mathbf{q}_e - \alpha_2^T k_4 \alpha_2 \leq 0 \tag{3.38}$$

3.3 New system

After the complete backstepping, our new system becomes:

$$\dot{\eta} = \begin{bmatrix} \dot{\mathbf{e}}_{\xi,1} \\ \dot{\mathbf{e}}_{\xi,2} \\ \dot{\mathbf{q}}_e \\ \dot{\alpha}_2 \end{bmatrix} = \begin{bmatrix} -\mathbf{a}.\bar{\mathbf{e}}_{\xi,2} - k_1\bar{\mathbf{e}}_{\xi,1} \\ \mathbf{q} \otimes \frac{F_u}{m} \otimes \mathbf{q}^* + F_{ext} - (\frac{k_1\dot{\bar{\mathbf{e}}}_{\xi,1}}{a} + \ddot{\xi}_{target}) \\ \frac{1}{2}M(\mathbf{q}_{\bar{e}})(\Omega^v + \alpha_2) \\ I^{-1}(-\Omega \times I\Omega + \tau) - \dot{\Omega}^v \end{bmatrix} \tag{3.39}$$

The control inputs are written in equation 3.23 and 3.37.

Chapter 4

Numerical Simulations

In this chapter, the proposed control will be applied into the nonlinear system. We will use a constant reference point and we will follow a predefined desired trajectory such as an ascending spiral trajectory, a lemniscate trajectory in 3D, and a cylinder trajectory. These tests have some hyperparameters that we can use to alter the performance of the drone. We can change the initial conditions close and far to the origin, we can increase the gain while keeping it positive, we can add an offset distance to make sure the agent stays at a safe distance with respect to the trajectory. We will also introduce some noise to the states to test the robustness of the model to environment noise, mainly due to sensor limitations.

4.1 Convergence to a Fixed Reference Point

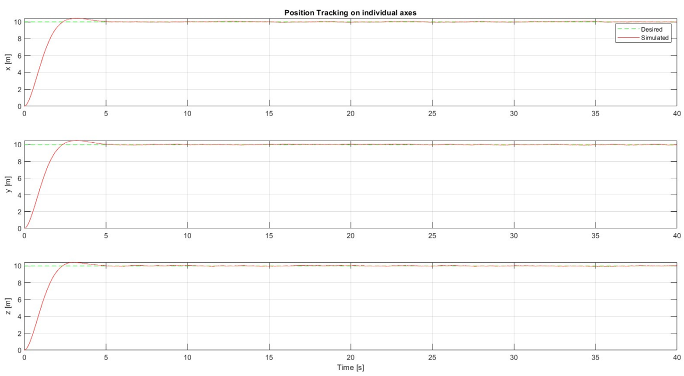

Figure 4.1: Target Convergence Position Evolution

The system converges to $\xi_d = (10, 10, 10)^T$ in 5 seconds.

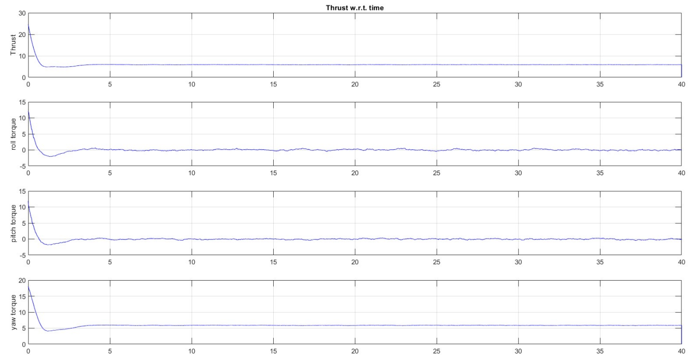

Figure 4.2: Target Convergence Control Input Evolution

Figure 4.2 shows a reasonable stability for the control inputs that won't change since the trajectory is a straight line.

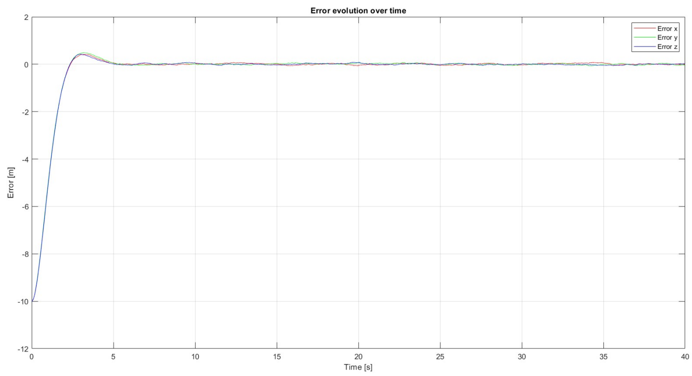

Figure 4.3: Target Convergence Errors Evolution

The error variables that we set in the new system converge to 0 which aligns with the proposed Lyapunov function and validates the stability of the main variables. Thus, convergence is attained in 5 seconds.

4.2 Following a 3D trajectory

In this section, we want to test the performance of our control law following a more complex trajectory.

We have the option to consider a 0 yaw value throughout our experiments or consider that the drone has a proprioceptive sensor in front along the longitudinal axis. The latter suggests that the yaw should share the same direction as the horizontal velocity vector.

TO make the movement more realistic, we choose option 2 and represent the body yaw angle by an arrow in our animations.

4.2.1 Cylindrical

Equation

Let's define the 3D cylinder trajectory:

$$\mathbf{x}_d(t) = \begin{bmatrix} R\cos(\omega t) \\ R\sin(\omega t) \\ z_0 + v_z t \end{bmatrix}, \qquad t \in [0, T] \tag{4.1}$$

where the parameters are defined as:

$$R \in \mathbb{R}^+ \qquad \text{radius of the cylinder} \tag{4.2}$$

$$\omega \in \mathbb{R}^+ \qquad \text{angular velocity around the cylinder} \tag{4.3}$$

$$z_0 \in \mathbb{R} \qquad \text{initial altitude} \tag{4.4}$$

$$z_f \in \mathbb{R} \qquad \text{final altitude} \tag{4.5}$$

$$T \in \mathbb{R}^+ \qquad \text{trajectory duration} \tag{4.6}$$

$$v_z = \frac{z_f - z_0}{T} \qquad \text{constant vertical velocity} \tag{4.7}$$

This function is used to simulate the drone's trajectory following capabilities in missions that require patrolling and surveillance.

Application 1

This trial has the following parameters:

- gains = 1;
- model noise;

- initialization close the initial desired point.

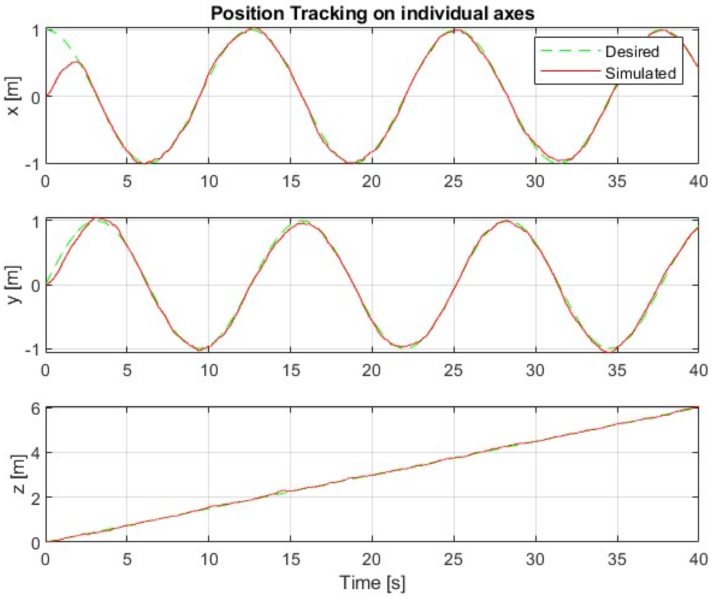

Figure 4.4: Cylindrical Trajectory Following - Position Evolution

The system catches the trajectory in a speedy way ($\approx 2s$). We can see in figure 4.5 that the noise applied to the state vector affects the roll and pitch torques.

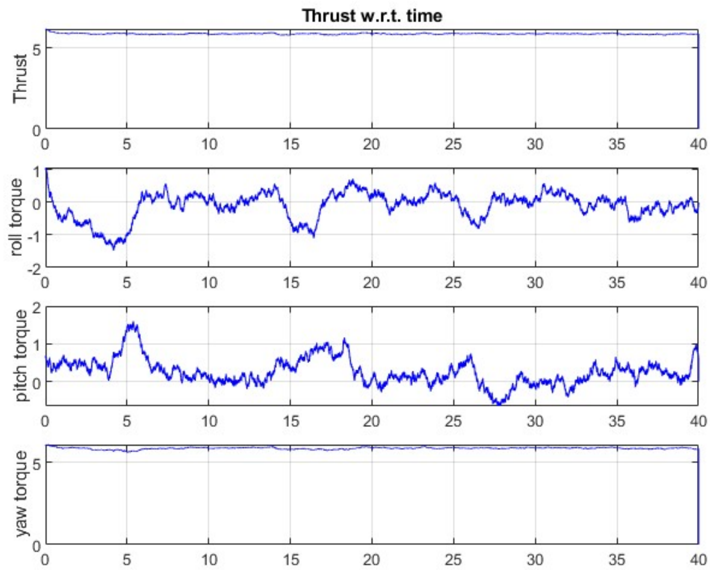

Figure 4.5: Cylindrical Trajectory Following - Control Input Evolution

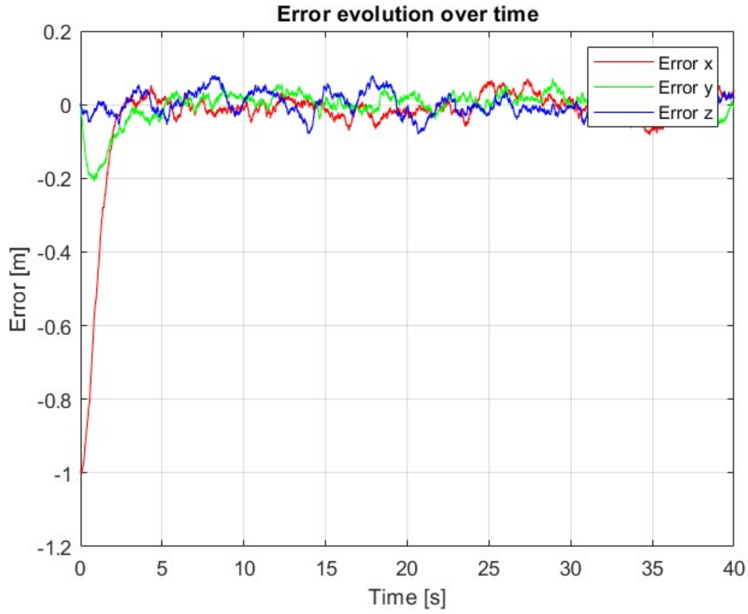

Figure 4.6: Cylindrical Trajectory Following - Errors Evolution

The errors converge to 0 in a noisy manner.

Application 2

In this trial, we apply a uniform distribution to the gains $k_i \forall i = 1, 2, 3, 4$. We predict that this attenuation of the gains will slow down the convergence and require a greater thrust to compensate.

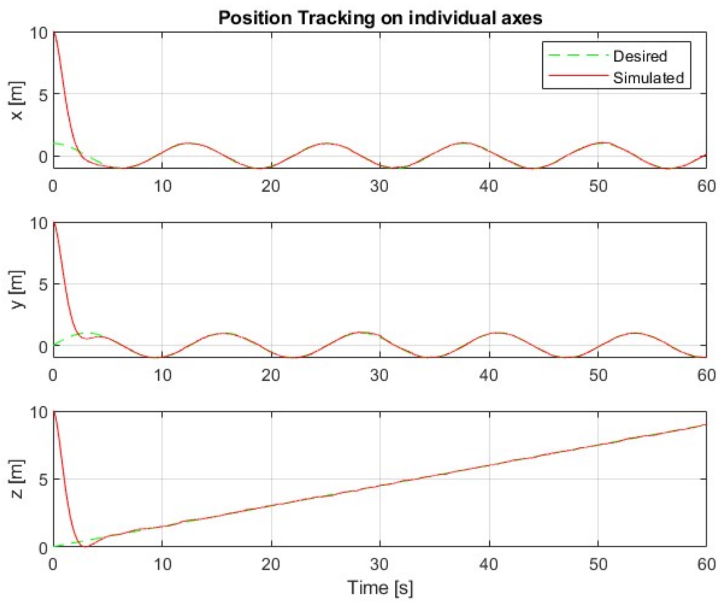

Figure 4.7: Cylindrical Trajectory Following - Position Evolution

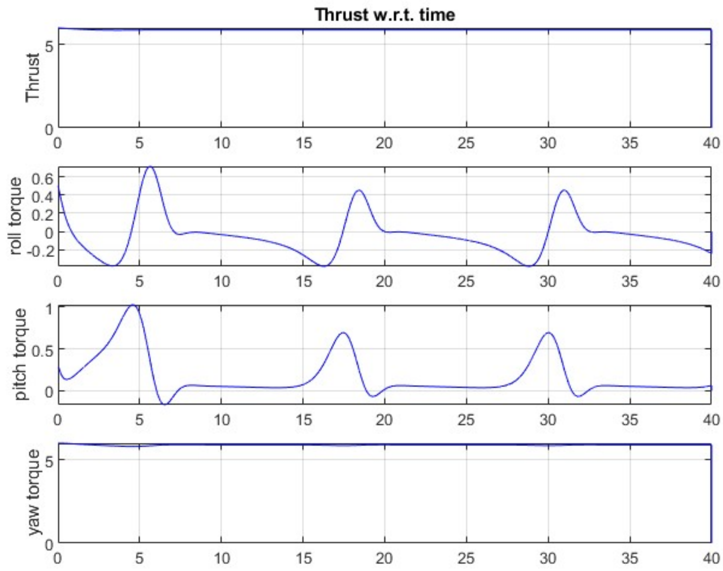

Figure 4.8: Cylindrical Trajectory Following - Control Input Evolution

Figure 4.8 shows a constant yaw value which is very logical for a cylindrical trajectory because the drone is moving in a circular manner at constant speed.

The roll and pitch torques have spikes and then stabilize each time the drone turns and accelerates.

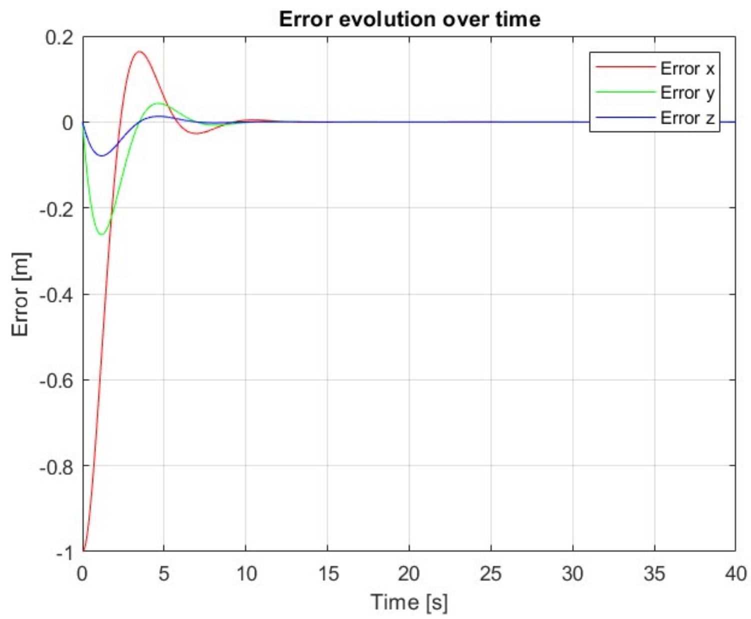

Figure 4.9: Cylindrical Trajectory Following - Errors Evolution

4.2.2 Spiral

The 3D spiral trajectory brings an innovation by having a variable radius compared to the cylindrical trajectory.

Equation

Let's define the equation of ascending spiral trajectory:

$$\mathbf{x}_d(t) = \begin{bmatrix} (R_0 + v_r\, t)\cos(\omega t) \\ (R_0 + v_r\, t)\sin(\omega t) \\ z_0 + v_z\, t \end{bmatrix}, \qquad t \in [0, T] \tag{4.8}$$

where the parameters are defined as:

$$R_0 \in \mathbb{R}^+ \qquad \text{initial radius of the spiral} \tag{4.9}$$

$$R_f \in \mathbb{R}^+ \qquad \text{final radius of the spiral} \tag{4.10}$$

$$\omega \in \mathbb{R}^+ \qquad \text{angular velocity} \tag{4.11}$$

$$z_0 \in \mathbb{R} \qquad \text{initial altitude} \tag{4.12}$$

$$z_f \in \mathbb{R} \qquad \text{final altitude} \tag{4.13}$$

$$T \in \mathbb{R}^+ \qquad \text{trajectory duration} \tag{4.14}$$

$$v_r = \frac{R_f - R_0}{T} \qquad \text{radial expansion velocity} \tag{4.15}$$

$$v_z = \frac{z_f - z_0}{T} \qquad \text{constant vertical velocity} \tag{4.16}$$

Application

This trial has the following parameters:

- gains = 1;
- model noise;
- initialization close the initial desired point.

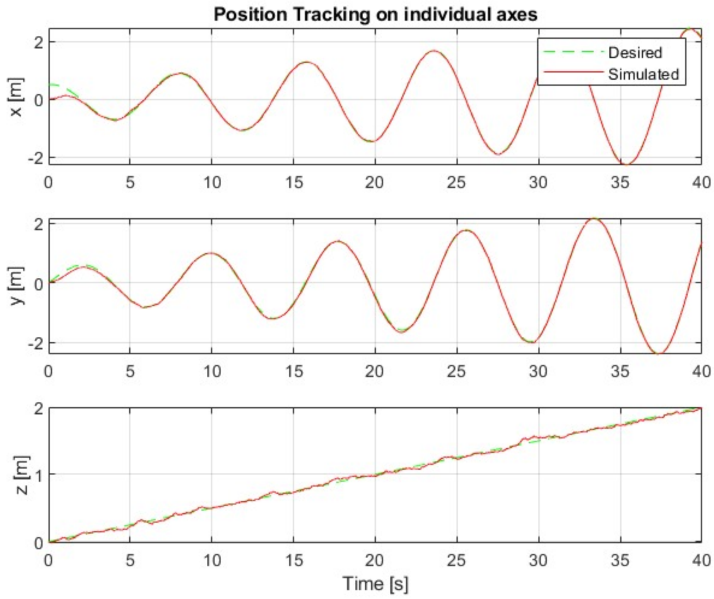

Figure 4.10: Spiral Trajectory Following - Position Evolution

Our system behaves in a similar fashion with respect to the previous case with the cylindrical trajectory.

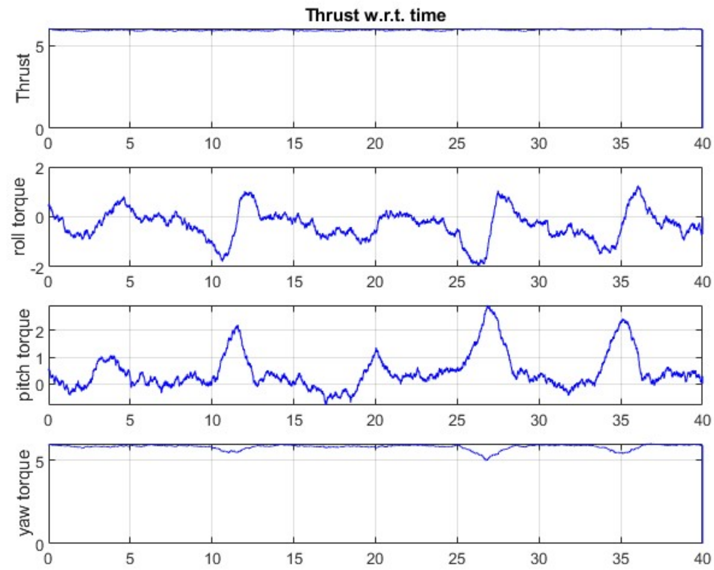

Figure 4.11: Spiral Trajectory Following - Control Input Evolution

The roll and pitch torques share a similar periodicity with the previous test. However with the addition of the model noise, the drone is constantly trying to maintain its balance.

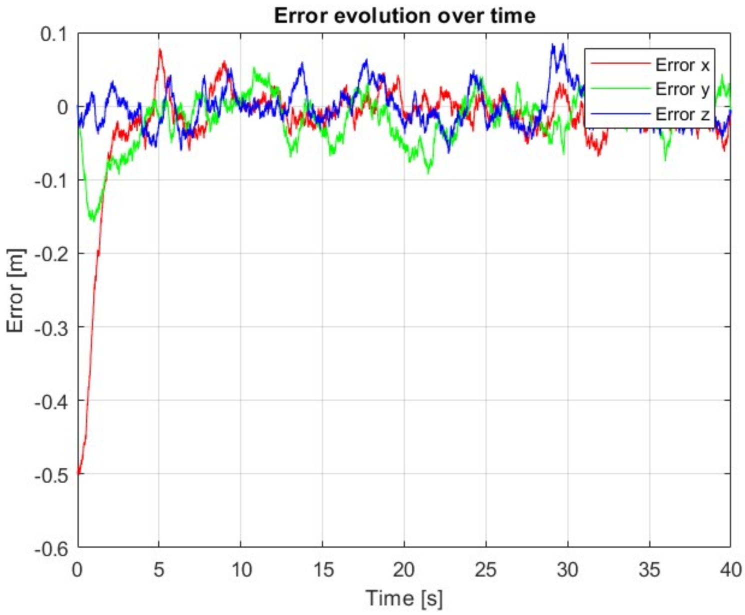

Figure 4.12: Spiral Trajectory Following - Errors Evolution

4.2.3 Lemniscate

Equation

Let's define the lemniscate trajectory in 3D

$$\mathbf{x}_d(t) = \begin{bmatrix} \dfrac{A\cos(\omega t)}{1+\sin^2(\omega t)} \\ \dfrac{A\sin(\omega t)\cos(\omega t)}{1+\sin^2(\omega t)} \\ z_0 + v_z t \end{bmatrix}, \qquad t \in [0, T] \tag{4.17}$$

where the parameters are defined as:

$$A \in \mathbb{R}^+ \qquad \text{amplitude (semi-major axis of the lemniscate)} \tag{4.18}$$

$$\omega \in \mathbb{R}^+ \qquad \text{angular velocity} \tag{4.19}$$

$$z_0 \in \mathbb{R} \qquad \text{initial altitude} \tag{4.20}$$

$$z_f \in \mathbb{R} \qquad \text{final altitude} \tag{4.21}$$

$$T \in \mathbb{R}^+ \qquad \text{trajectory duration} \tag{4.22}$$

$$v_z = \frac{z_f - z_0}{T} \qquad \text{constant vertical velocity} \tag{4.23}$$

Application 1

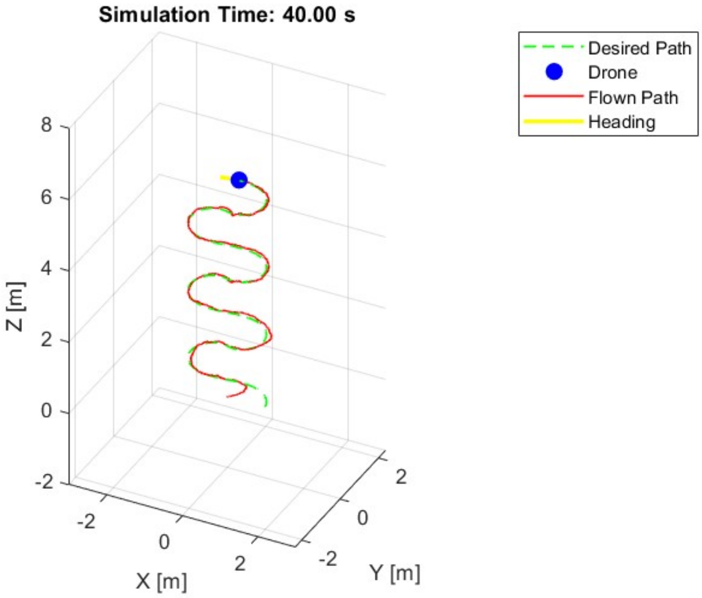

Figure 4.13: Lemniscate Trajectory Following - 3D Position Evolution - Near

It is clear in figure 4.13 that the lemniscate provides the biggest challenge yet since it changes directions more frequently. Our control system performs well and follows the trajectory to perfection.

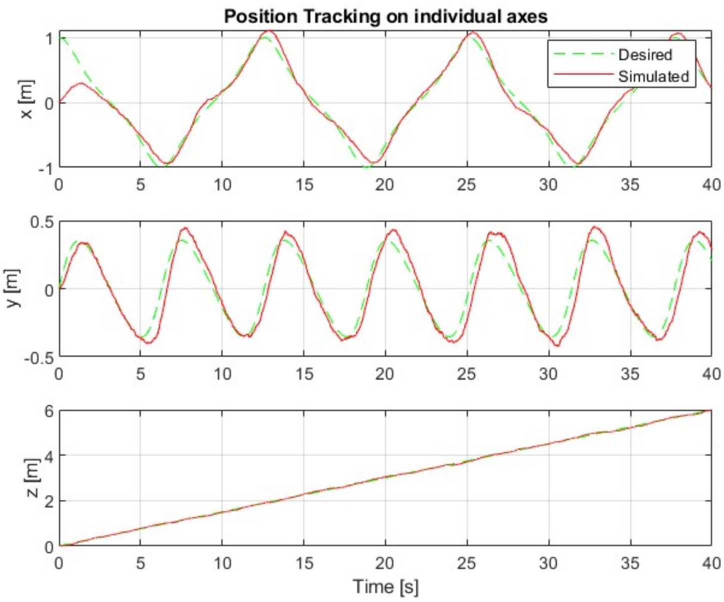

Figure 4.14: Lemniscate Trajectory Following - Position Evolution - Near

The triangular evolutions for the longitudinal and lateral axes is interesting. They represent the swift change for the x-axis and the smooth change for the y-axis.

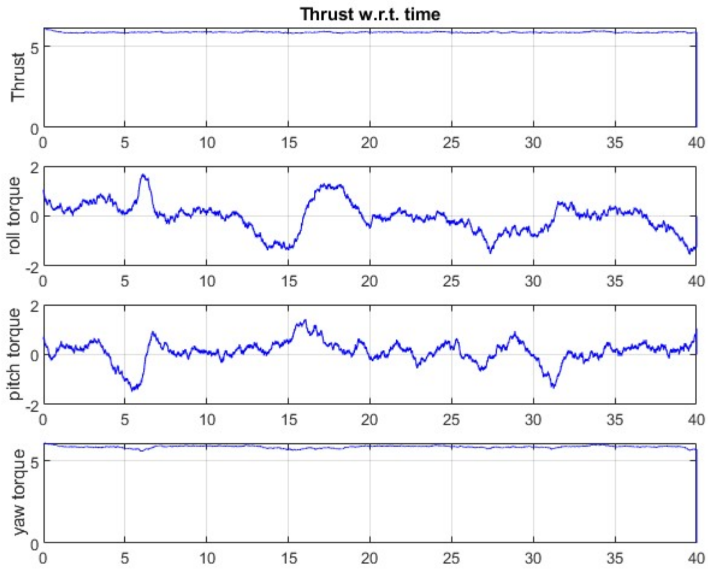

Figure 4.15: Lemniscate Trajectory Following - Control Input Evolution - Near

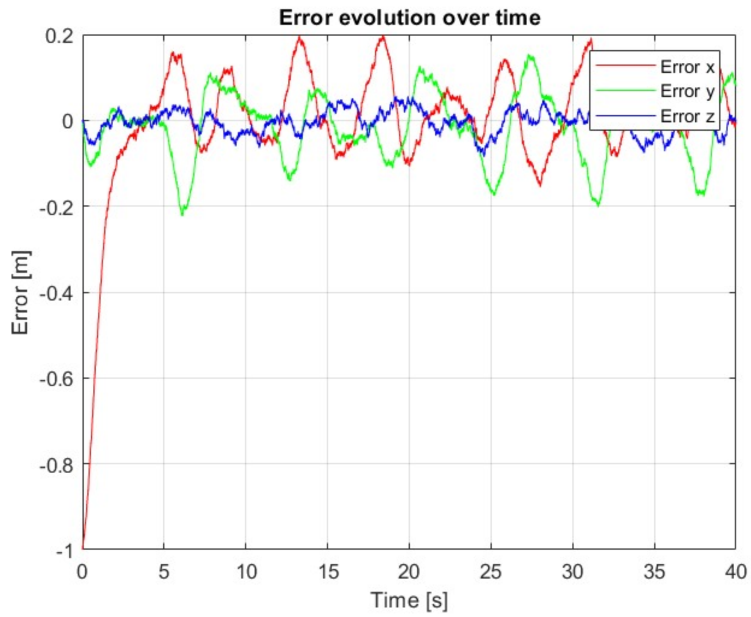

Figure 4.16: Lemniscate Trajectory Following - Errors Evolution - Near

The magnitude of the error variables is the biggest we had yet mainly due to steady state error. The trajectory does not provide any stable sections and the constant movement yields this offset. However it is not much and it is clearly justified.

Application 2

This test mirros the previous one but with a further initialization point.

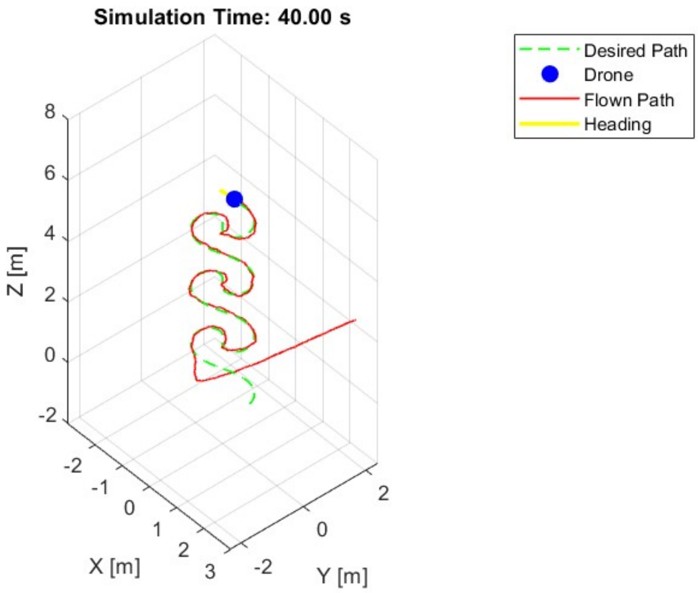

Figure 4.17: Lemniscate Trajectory Following - 3D Position Evolution - Far

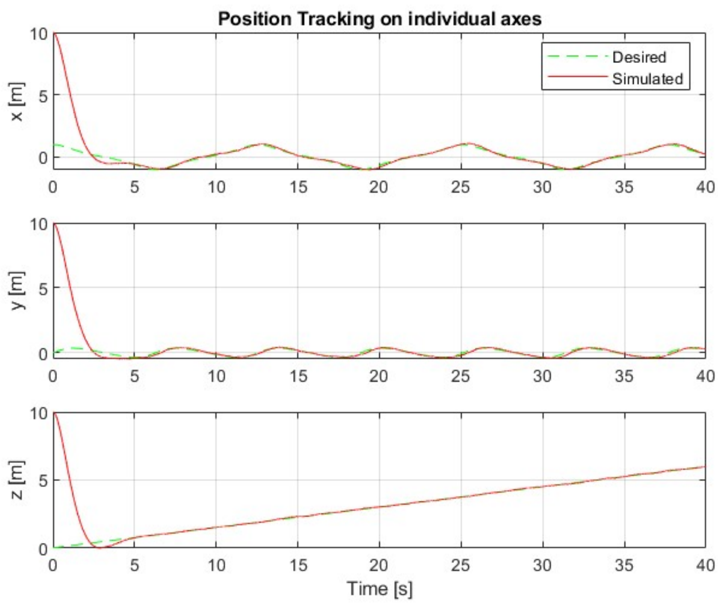

Figure 4.18: Lemniscate Trajectory Following - Position Evolution - Far

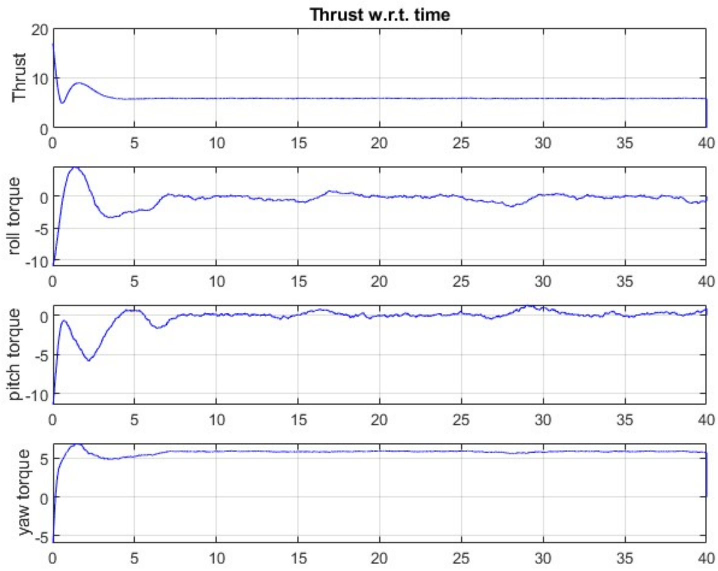

Figure 4.19: Lemniscate Trajectory Following - Control Input Evolution - Far

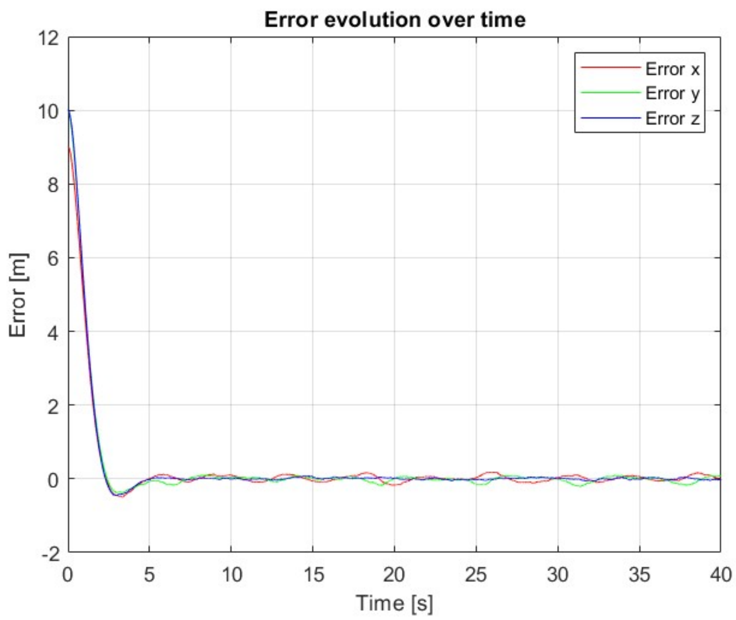

Figure 4.20: Cylindrical Trajectory Following - Errors Evolution - Far

Figures 4.18, 4.19, and 4.20 show the same performance as before after stabilising. The initial point is the only reason that the control inputs' values rise at the start.

4.3 Multi-Agent Coordination

Our system is very modular and along with the addition of new agents, we can easily update it to have a multi-agent coordination task. In this case, the defined task is to follow the leader in a platoon manner based on consensus control. Agent 2 receives the information of agent 1 and agent 3 from agent 2.

The control law is updated by changing the tracking error function of the variable e_1 to have it interfacing with other agents.

We need to define an offset matrix to make sure a safe distance is maintained at all times. Additionally, we can even define the Laplacian to showcase the communication between the agents.

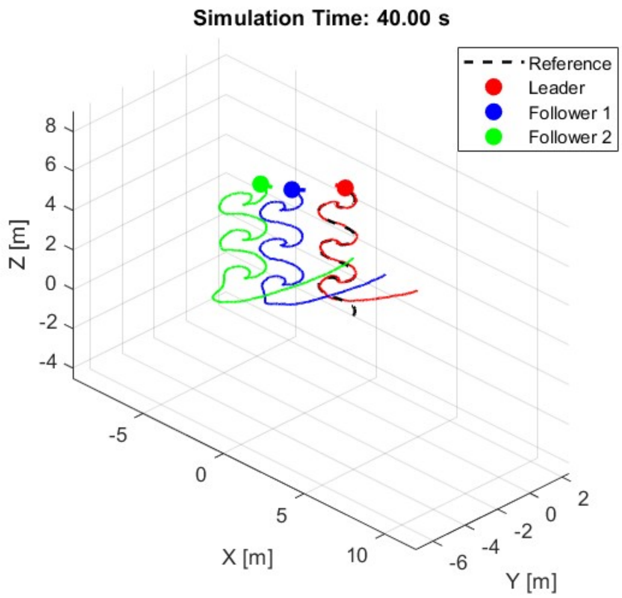

Figure 4.21: 3 Agents Following the Lemniscate Trajectory

4.4 Animations

Kindly refer to the below link to access all relevant media that showcase the result of our simualtions. They include trials for fixed reference point, Cylindrical trajectory following, Spiral trajectory following, Leminscate trajectory following, and Multi-Agent Coordination following a leminscate trajectory. Media and Simulations

Chapter 5

Discussions

The results align well with the stability study conducted in the control-theory part. The simulation results validate our knowledge about the backstepping control and quadcopter modeling using the quaternion method. The choice of the quaternion method proved crucial to avoid any potential singularities.

Chapter 6

Conclusion

This project allowed us to control an aerial mobile robot using quaternions and covering all considerations. The performance as well as the analysis that we went through were crucial to better grasp the concepts taught in class.

6.1 Opinions

The backstepping methodology is a clear guide that helps us to control non-linear systems through Lyapunov functions and thorugh studies. Moreover, we chose the quaternion approach for 2 reasons. The first one was to avoid singularities such as gimball lock and the second reason was to get to apply what seamed very complex and advanced in class. Once the equations started to fit together, the problem became easier to solve.

6.2 Future Work

Our future works focuses on the enhancement and implementation of the system through:

- Sensor-Based Control
- Fl-air and ROS implementation
- Hardware choice and integration
- Comparison of backstepping control with other control methods (NLMPC,AI, etc.)

Appendix A

MATLAB Source Code

Listing A.1: Main

```
%% ARS5 Project
% Title:     Quadcopter Backstepping Controller
%
% Authors:   Stefano MUSSO PIANTELLI,
%            Elias Charbel SALAMEH
%
% Date:      16/01/2026

clear;
clc;
close all;

% Physical parameters
g        = 9.81;
m        = 0.6;
Ixx      = 0.0014; Iyy = 0.0165; Izz = 0.0152;
I        = diag([Ixx Iyy Izz]);
F_ext    = [-0.0*randn(1,1);-0.1*randn(1,1);-g];

% Simulation parameters
Tend     = 40;
```

```
dt       = 1e-3;         % smaller dt for better attitude dynamics
    resolution
t        = 0:dt:Tend;
N        = numel(t);


%run('Trajectory_spiral.m');
run('Trajectory_lemniscate.m');
%run('Trajectory_cylindrical.m');
%run('Target_point.m');

%% State vector: X = [xi(3); v(3); q(4); Omega(3)]
% quaternion q = [qw; qx; qy; qz], unit norm
X = zeros(13, N);

% Initial conditions:
xi0      = [0; 0; 0];                  % initial position
v0       = [0; 0; 0];                  % initial velocity
q0       = [1; 0; 0; 0];               % initial attitude (no rotation)
Omega0   = [0; 0; 0];                  % initial angular velocity
X(:,1)   = [xi0; v0; q0; Omega0];

%% Gains
% k1   = rand(1);
% k2   = rand(1);
% k3   = rand(1);
% k4   = rand(1);
k1   = 1;
k2   = 1;
k3   = 1;
k4   = 1;

a    = 1;

%Offset
```

```
d    = 0.0;

%% New system after the control law
e1 = zeros(3, N);
e1_dot = zeros(3, N);
e2 = zeros(3, N);
e2_dot = zeros(3, N);
q_e = zeros(4, N);
alpha_2 = zeros(3, N);

xi_v_dot = zeros(3,N);
xi_v_ddot = zeros(3,N);
omega_v = zeros(3,N);
omega_v_dot = zeros(3,N);
q_e_dot = zeros(4,N);
alpha_2_dot = zeros(3,N);
Fu = zeros(3,N);
psi_d = zeros(1,N);
tau = zeros(3,N);
F_des_norm = zeros(1,N);

% sampled loop
for k = 1:N-1
    tk = t(k);

    xi_k = X(1:3,k);
    xi_dot_k = X(4:6,k);
    q_k = X(7:10,k);
    omega_k = X(11:13,k);

    % target, first and second derivate
    xi_d_val      = xi_d(tk);
    xi_d_dot_val  = xi_d_dot(tk);
    xi_d_ddot_val = xi_d_ddot(tk);

```

```
    % variable yaw logic
    % Calculate desired heading based on planar velocity
    vx_des = xi_d_dot_val(1);
    vy_des = xi_d_dot_val(2);
    horizontal_speed = sqrt(vx_des^2 + vy_des^2);

    if horizontal_speed > 0.1 % Threshold (0.1 m/s)
        % if moving: look where we are going
        psi_d(k) = atan2(vy_des, vx_des);
    else
        % if hovering or vertical: ignore new
        if k > 1
            psi_d(k) = psi_d(k-1);
        else
            psi_d(k) = 0; % init
        end
    end
    % ---

    % Translational control part
    e1(:,k) = -(a*(xi_k - xi_d_val)-d);
    xi_v_dot(:,k) = (k1*e1(:,k))/a + xi_d_dot_val;
    e2(:,k) = xi_dot_k - xi_v_dot(:,k);
    e1_dot(:,k) = -a*e2(:,k)-k1*e1(:,k);
    % F_des: control input
    vec1 = m * ( (a * e1(:,k)) - F_ext + ...
                 ((k1 * e1_dot(:,k)) / a + xi_d_ddot_val) - ...
                 (k2 * e2(:,k)) );

    q_row = q_k.';
    q_left = quatinv(q_row);
    vec_quat = [0, vec1'];

    %compute F_u
    F_des_quat = quatmultiply(quatmultiply(q_left, vec_quat), q_row);
```

```
    F_des = F_des_quat(2:4)';

    Fu(:,k) = F_des;
    F_des_norm(:,k) = norm(F_des);


    vec2 = Fu(:,k)/m;
    q_conj2 = quatconj(q_row);
    vec_quat2 = [0, vec2'];
    e2_dot_quat = quatmultiply(quatmultiply(q_row, vec_quat2), q_conj2)
        ...
                    + F_ext - (k1*e1_dot(:,k)/a + xi_d_ddot_val);
    e2_dot(:,k) = e2_dot_quat(2:4); % e2dot

    % attitude extraction
    if F_des_norm(:,k) < 1e-6
        % Fallback: Zero thrust and identity orientation (or keep
            previous)
        Fu_des_norm = 0;
        q_d = [1, 0, 0, 0]; % Identity quaternion [w x y z]
        warning('Fu magnitude is near zero. Thrust set to 0, attitude
            set to identity.');
        % tau = 0;
    end

    z_d = F_des / F_des_norm(:,k);
    %Temporal axis
    l_vec = [-sin(psi_d(:,k)); cos(psi_d(:,k)); 0];

    l_cross_zd = cross(l_vec, z_d);
    norm_l_cross_zd = norm(l_cross_zd);
    if norm_l_cross_zd < 1e-6
        if abs(z_d(3)) < 0.999
            temp_v = [0; 0; 1];
            x_d_temp = cross(temp_v, z_d);
```

```
                x_d = x_d_temp / norm(x_d_temp);
            else
                x_d = [1; 0; 0];
            end
            warning('Singularity detected in attitude extraction (l
               parallel to z_d). Using fallback x_d.');
        else
            %x axis
            x_d = l_cross_zd / norm_l_cross_zd;
        end
        %y axis
        y_d = cross(z_d, x_d);
        %Rotational matrix desired
        Rd = [x_d, y_d, z_d];
        q_d = rotm2quat(Rd);


        % Rotational control part
        % quaternion error
        q_k   = q_k / norm(q_k);
        q_d = q_d / norm(q_d);
        q_d_conj = quatconj(q_d);
        q_bar_e  = quatmultiply(q_d_conj, q_k');
        q_bar_e0 = q_bar_e(1); q_bar_ev = q_bar_e(2:4).';
        q_e(:,k) = [1 - abs(q_bar_e0); q_bar_ev];

        % M : quaternion error dynamics computation
        M = quatM(q_bar_e);

        omega_v(:,k) = -2*k3*M.'*q_e(:,k);
        alpha_2(:,k) = omega_k - omega_v(:,k);
        q_e_dot(:,k) = 0.5 * M * (alpha_2(:, k) + omega_v(:,k));
        M_dot = quatM(q_e_dot(:,k));
        omega_v_dot = -2*k3*(M_dot.' * q_e(:,k) + M.' * q_e_dot(:,k));

```

```
    %Tau
    tau = cross(omega_k,I*omega_k)+I*omega_v_dot-k4*alpha_2(:, k)-0.5*M
        .'*q_e;
    alpha_2_dot(:,k) = I\(-cross(omega_k,I*omega_k)+tau(:, k))-
        omega_v_dot;


    xi_dot = xi_dot_k;

    vec_rotated = quatmultiply(quatmultiply(q_row, vec_quat2), q_conj2)
        ;  % 1x4
    xi_ddot = vec_rotated(2:4).' + F_ext;  % 3x1
    q_dot = 0.5*quatmultiply(q_k',[0 (omega_k.')]);

    omega_dot = I\(tau-cross(omega_k,I*omega_k));

    % Noise
    noise_pos   = 0.001 * randn(3,1);    % Position noise (usually 0
        for physics, this is mostly sensor noise)
    noise_vel   = 0.001 * randn(3,1);    % Velocity noise (Wind gusts,
        drag irregularities) - in m/s
    noise_q     = 0.001 * randn(4,1);    % Attitude noise (Vibrations
        affecting integration)
    noise_omega = 0.001 * randn(3,1);    % Angular velocity noise (
        Torque disturbances) - in rad/s

    % System update with computed control
    X(1:3,k+1)   = xi_k       + dt * xi_dot + noise_pos;
    X(4:6,k+1)   = xi_dot_k   + dt * xi_ddot + noise_vel;

    q_next       = q_k        + dt * q_dot.' + noise_q;
    X(7:10,k+1)  = q_next / norm(q_next);

    X(11:13,k+1) = omega_k    + dt * omega_dot(:, k) + noise_omega;
end
```

```

%% computation of the desired position xi
xi_d_all = zeros(3, N);
for k = 1:N
    xi_d_all(:,k) = xi_d(t(k));
end

%% Plots
xi_sim = X(1:3, :);

% 3D position evolution
figure('Name', '3D Position', 'Color', 'w');
plot3(xi_d_all(1,:), xi_d_all(2,:), xi_d_all(3,:), 'g--', 'LineWidth',
    1.5); hold on;
plot3(xi_sim(1,:), xi_sim(2,:), xi_sim(3,:), 'r', 'LineWidth', 1.5);
grid on; axis equal;
xlabel('X [m]'); ylabel('Y [m]'); zlabel('Z [m]');
legend('Desired (Spiral)', 'Simulated (Backstepping)');
title('3D Trajectory Tracking');
view(45, 30);

% Temporal analysis for each axis (x, y, z)
figure('Name', 'Axis Analysis x, y, z', 'Color', 'w');
subplot(3,1,1);
plot(t, xi_d_all(1,:), 'g--', t, xi_sim(1,:), 'r');
ylabel('x [m]'); grid on;
legend('Desired', 'Simulated');
title('Position Tracking on individual axes');

subplot(3,1,2);
plot(t, xi_d_all(2,:), 'g--', t, xi_sim(2,:), 'r');
ylabel('y [m]'); grid on;

subplot(3,1,3);
plot(t, xi_d_all(3,:), 'g--', t, xi_sim(3,:), 'r');
```

```
ylabel('z [m]'); xlabel('Time [s]'); grid on;

% Position Error Plot
error_pos = xi_sim - xi_d_all;
figure('Name', 'Tracking Error', 'Color', 'w');
plot(t, error_pos(1,:), 'r', t, error_pos(2,:), 'g', t, error_pos(3,:),
    'b');
grid on;
xlabel('Time [s]'); ylabel('Error [m]');
legend('Error x', 'Error y', 'Error z');
title('Error evolution over time');

% control input applied
figure('Name', 'Control Inputs', 'Color', 'w');
subplot(4,1,1);
plot(t, F_des_norm(1,:), 'b');
ylabel('Thrust'); grid on;
title('Thrust w.r.t. time');

subplot(4,1,2);
plot(t, Fu(1,:), 'b');
ylabel('roll torque'); grid on;

subplot(4,1,3);
plot(t, Fu(2,:), 'b');
ylabel('pitch torque'); grid on;

subplot(4,1,4);
plot(t, Fu(3,:), 'b');
ylabel('yaw torque'); grid on;



%% 3D Animation + Video
fprintf('Starting 3D Animation...\n');
```

```

% Animation Parameters
anim_speed  = 100;      % Plot every anim_speed steps
scale_arrow = 0.5;      % Length of heading arrow

% FIGURE
figure('Name', '3D Drone Animation', 'Color', 'w');
axis equal;
grid on;
hold on;
view(45, 30);
xlabel('X [m]'); ylabel('Y [m]'); zlabel('Z [m]');
title('Quadcopter Trajectory Animation');

% Limits (keep camera stable)
xlim([min(xi_d_all(1,:))-2, max(xi_d_all(1,:))+2]);
ylim([min(xi_d_all(2,:))-2, max(xi_d_all(2,:))+2]);
zlim([min(xi_d_all(3,:))-2, max(xi_d_all(3,:))+2]);

% Desired trajectory (background)
plot3(xi_d_all(1,:), xi_d_all(2,:), xi_d_all(3,:), ...
      'g--', 'LineWidth', 1, 'DisplayName', 'Desired Path');

% GRAPHICS OBJECTS
% Drone body
h_drone = plot3(0, 0, 0, 'bo', ...
    'MarkerFaceColor', 'b', 'MarkerSize', 8, 'DisplayName', 'Drone');

% Trail
h_trail = plot3(0, 0, 0, 'r-', ...
    'LineWidth', 1, 'DisplayName', 'Flown Path');

% Heading arrow (Body-X axis)
h_arrow = line([0 0], [0 0], [0 0], ...
    'Color', 'y', 'LineWidth', 2, 'DisplayName', 'Heading');
```

```

legend('show');

% VIDEO WRITER
video_name = 'Quadcopter_Animation.mp4';
v = VideoWriter(video_name, 'MPEG-4');
v.FrameRate = 30;      % frames per second
v.Quality   = 100;     % max quality
open(v);

%ANIMATION LOOP
for k = 1:anim_speed:N

    % Current state
    pos  = X(1:3, k);
    quat = X(7:10, k);   % [w x y z]

    % Drone position
    set(h_drone, 'XData', pos(1), 'YData', pos(2), 'ZData', pos(3));

    % Trail
    set(h_trail, 'XData', X(1,1:k), ...
                 'YData', X(2,1:k), ...
                 'ZData', X(3,1:k));

    % Heading arrow
    R_current   = quat2rotm(quat');       % Rotation matrix
    heading_vec = R_current(:,1);         % Body-X in world
    arrow_end   = pos + scale_arrow * heading_vec;

    set(h_arrow, 'XData', [pos(1), arrow_end(1)], ...
                 'YData', [pos(2), arrow_end(2)], ...
                 'ZData', [pos(3), arrow_end(3)]);

    % Title with time
```

```
    title(sprintf('Simulation Time: %.2f s', t(k)));

    % Render & save frame
    drawnow limitrate;
    frame = getframe(gcf);
    writeVideo(v, frame);

end

%CLOSE VIDEO
close(v);
fprintf('Video successfully saved: %s\n', video_name);
```

Listing A.2: Multi Agent

```
%% ARS5 Project
% Title:    Multi-Agent Platooning (Daisy-Chain)
% Authors:  Stefano MUSSO PIANTELLI, Elias Charbel SALAMEH
% Date:     15/01/2026

clear; clc; close all;

%% Parameters
num_agents = 3;

g = 9.81;
m = 0.6;
I = diag([0.0014, 0.0165, 0.0152]);

Tend = 40;
dt   = 1e-3;
t    = 0:dt:Tend;
N    = numel(t);

%% Desired trajectory Ascending spiral
% run('Trajectory_spiral.m');
run('Trajectory_lemniscate.m');
% run('Trajectory_cylindrical.m');
% run('Target_point.m');

%% State Initialization
X = zeros(13, N, num_agents);

% Robot 1: Leader
xi0_1 = [10; -5; 7];
% Robot 2: 2m behind Leader
xi0_2 = xi0_1 + [-2; 0; 0];
% Robot 3: 2m behind Robot 2
xi0_3 = xi0_2 + [-2; 0; 0];
```

```

xi0_all = [xi0_1, xi0_2, xi0_3];

for i = 1:num_agents
    X(:, 1, i) = [xi0_all(:,i); zeros(3,1); [1;0;0;0]; zeros(3,1)];
end

%% Platooning Topology (1 -> 2 -> 3)
A = zeros(num_agents, num_agents);
A(2,1) = 1; % Robot 2 listens to Robot 1
A(3,2) = 1; % Robot 3 listens to Robot 2 (NOT Robot 1)

% Desired Relative Offsets (xi_i - xi_j)_desired
% "Stay 2m behind the one you are following"
D_offset = zeros(3, num_agents, num_agents);
D_offset(:, 2, 1) = [-2; -1.5; -0.3]; % R2 relative to R1
D_offset(:, 3, 2) = [-2; -0.4; -0.3]; % R3 relative to R2

%% Gains
k1 = 1;
k2 = 1;
k3 = 1;
k4 = 1;
a  = 1;

%% Pre-allocation
e1     = zeros(3, N, num_agents);
e1_dot = zeros(3, N, num_agents);
e2     = zeros(3, N, num_agents);
Fu     = zeros(3, N, num_agents);
psi_d  = zeros(1, N, num_agents);

%% Main Loop
for k = 1:N-1
    tk = t(k);
```

```

    % Global Trajectory (Leader reference)
    xi_d_val      = xi_d(tk);
    xi_d_dot_val  = xi_d_dot(tk);
    xi_d_ddot_val = xi_d_ddot(tk);

    for i = 1:num_agents

        % State Extraction
        xi_i      = X(1:3, k, i);
        v_i       = X(4:6, k, i);
        q_i       = X(7:10, k, i);
        omega_i   = X(11:13, k, i);

        % Consensus Error
        sum_pos_error = [0;0;0];
        sum_vel_ref   = [0;0;0];
        sum_acc_ref   = [0;0;0];
        degree_connectivity = 0;

        % R1
        if i == 1
            sum_pos_error = (xi_i - xi_d_val);
            sum_vel_ref   = xi_d_dot_val;
            sum_acc_ref   = xi_d_ddot_val;
            degree_connectivity = 1;

        % R2, 3
        else
            % Iterate to find who I am following
            for j = 1:num_agents
                if A(i,j) == 1
                    xi_j = X(1:3, k, j);
                    v_j  = X(4:6, k, j);

```

```
                    % Error: xi_i-xi_k-d_ij
                    sum_pos_error = sum_pos_error + (xi_i - xi_j - D_offset(:, i, j));

                    % Feedforward Velocity: Copy his velocity
                    sum_vel_ref   = sum_vel_ref + v_j;

                    degree_connectivity = degree_connectivity + 1;
                end
            end
        end

        e1(:, k, i) = - (a * sum_pos_error);

        xi_v_dot = (sum_vel_ref + k1 * e1(:, k, i)) / max(1, degree_connectivity);

        e2(:, k, i) = v_i - xi_v_dot;
        e1_dot(:, k, i) = -a * e2(:, k, i) - k1 * e1(:, k, i);

        F_ext = [-0.1*randn;-0.1*randn;-g];

        vec1 = m * ( (a * e1(:,k,i)) - F_ext + ...
                     ((k1 * e1_dot(:,k,i))/a + sum_acc_ref) - ...
                     (k2 * e2(:,k,i)) );

        q_row  = q_i.';
        q_left = quatinv(q_row);
        vec_quat = [0, vec1'];
        F_des_quat = quatmultiply(quatmultiply(q_left, vec_quat), q_row);
        F_des = F_des_quat(2:4)';

        Fu(:, k, i) = F_des;
        F_des_norm = norm(F_des);
```

```

        % Platooning Yaw
        % Leader: Look at Velocity
        if i == 1
            vel_ref = xi_d_dot_val;
            if norm(vel_ref(1:2)) > 0.1
                psi_d(1, k, i) = atan2(vel_ref(2), vel_ref(1));
            else
                if k>1, psi_d(1, k, i) = psi_d(1, k-1, i); else, psi_d
                   (1, k, i) = 0; end
            end
        else
            % Follower: Look at Predecessor
            % Find 'j' where A(i,j)=1
            [~, predecessor_idx] = max(A(i,:));

            % Vector pointing to predecessor
            look_vec = X(1:3, k, predecessor_idx) - xi_i;

            if norm(look_vec(1:2)) > 0.1
                 psi_d(1, k, i) = atan2(look_vec(2), look_vec(1));
            else
                 if k>1, psi_d(1, k, i) = psi_d(1, k-1, i); else, psi_d
                    (1, k, i) = 0; end
            end
        end

        % Attitude Calculation
        z_d = F_des / max(F_des_norm, 1e-6);
        l_vec = [-sin(psi_d(1, k, i)); cos(psi_d(1, k, i)); 0];
        x_d = cross(l_vec, z_d); x_d = x_d/norm(x_d);
        y_d = cross(z_d, x_d);
        Rd = [x_d, y_d, z_d];
        q_d = rotm2quat(Rd);

```

```
        q_i = q_i / norm(q_i); q_d = q_d / norm(q_d);
        q_d_conj = quatconj(q_d);
        q_bar_e  = quatmultiply(q_d_conj, q_i');
        q_e = [1 - abs(q_bar_e(1)); q_bar_e(2:4).'];

        M = quatM(q_bar_e);
        omega_v = -2*k3*M.'*q_e;
        alpha_2 = omega_i - omega_v;
        q_e_dot = 0.5 * M * (alpha_2 + omega_v);
        M_dot = quatM(q_e_dot);
        omega_v_dot = -2*k3*(M_dot.' * q_e + M.' * q_e_dot);

        tau = cross(omega_i, I*omega_i) + I*omega_v_dot - k4*alpha_2 -
            0.5*M.'*q_e;

        % Integration
        vec2 = F_des/m;
        q_conj2 = quatconj(q_row);
        vec_quat2 = [0, vec2'];

        vec_rotated = quatmultiply(quatmultiply(q_row, vec_quat2),
            q_conj2);
        xi_ddot = vec_rotated(2:4).' + F_ext;

        q_dot = 0.5*quatmultiply(q_i', [0, omega_i']);
        omega_dot = I \ (tau - cross(omega_i, I*omega_i));

        noise_pos = 0.001 * randn(3,1); noise_vel = 0.001 * randn(3,1);

        X(1:3, k+1, i)   = xi_i     + dt * v_i + noise_pos;
        X(4:6, k+1, i)   = v_i      + dt * xi_ddot + noise_vel;
        q_next           = q_i      + dt * q_dot.';
        X(7:10, k+1, i)  = q_next / norm(q_next);
        X(11:13, k+1, i) = omega_i  + dt * omega_dot;
    end
```

```
end

%% Visualization (Platooning)
xi_d_all = zeros(3, N);
for k = 1:N, xi_d_all(:,k) = xi_d(t(k)); end

colors = {'r', 'b', 'g'};
labels = {'Leader', 'Follower 1', 'Follower 2'};

figure('Name', 'Platoon Trajectory', 'Color', 'w');
plot3(xi_d_all(1,:), xi_d_all(2,:), xi_d_all(3,:), 'g--', 'LineWidth',
    1.5, 'DisplayName', 'Reference');
hold on; grid on; axis equal; view(45, 30);
xlabel('X'); ylabel('Y'); zlabel('Z');

for i = 1:num_agents
    pos = squeeze(X(1:3, :, i));
    plot3(pos(1,:), pos(2,:), pos(3,:), 'Color', colors{i}, 'LineWidth'
        , 1.5, 'DisplayName', labels{i});
    plot3(pos(1,end), pos(2,end), pos(3,end), 's', 'Color', colors{i},
        'MarkerFaceColor', colors{i});
end
legend('show');
title('Platooning (Daisy-Chain) Control');

%% 3D Animation + Video Export (MULTI-ROBOT)
fprintf('Starting 3D Animation...\n');

anim_speed  = 100;
scale_arrow = 0.6;
margin      = 2;    % extra space around trajectories

%% ===== PRECOMPUTE GLOBAL AXIS LIMITS (ALL ROBOTS) =====
all_pos = reshape(X(1:3,:,:), 3, []);
xmin = min(all_pos(1,:)) - margin;
```

```
xmax = max(all_pos(1,:)) + margin;
ymin = min(all_pos(2,:)) - margin;
ymax = max(all_pos(2,:)) + margin;
zmin = min(all_pos(3,:)) - margin;
zmax = max(all_pos(3,:)) + margin;

%% ===== FIGURE =====
figure('Name', '3D Platoon Animation', 'Color', 'w');
axis equal; grid on; hold on;
view(45,30);
xlabel('X [m]'); ylabel('Y [m]'); zlabel('Z [m]');
title('Multi-Agent Platooning Animation');

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

%% ===== REFERENCE TRAJECTORY =====
h_ref = plot3(xi_d_all(1,:), xi_d_all(2,:), xi_d_all(3,:), ...
              'k--', 'LineWidth', 1.5);

%% ===== GRAPHICS OBJECTS =====
colors  = {'r','b','g'};
labels  = {'Leader','Follower 1','Follower 2'};

h_drone = gobjects(num_agents,1);
h_trail = gobjects(num_agents,1);
h_arrow = gobjects(num_agents,1);

for i = 1:num_agents
    h_drone(i) = plot3(0,0,0,'o', ...
        'Color', colors{i}, ...
        'MarkerFaceColor', colors{i}, ...
        'MarkerSize', 8);

```

```
    h_trail(i) = plot3(0,0,0,'-', ...
        'Color', colors{i}, ...
        'LineWidth', 1.2);

    h_arrow(i) = line([0 0],[0 0],[0 0], ...
        'Color', colors{i}, ...
        'LineWidth', 2);
end

%% ===== CORRECT LEGEND (USING HANDLES) =====
legend([h_ref; h_drone], ...
       [{'Reference'}, labels], ...
       'Location','best');

%% ===== VIDEO WRITER =====
video_name = 'Platooning_Animation.mp4';
v = VideoWriter(video_name,'MPEG-4');
v.FrameRate = 30;
v.Quality   = 100;
open(v);

%% ===== ANIMATION LOOP =====
for k = 1:anim_speed:N

    for i = 1:num_agents
        pos  = X(1:3, k, i);
        quat = X(7:10, k, i);

        % Drone body
        set(h_drone(i), ...
            'XData', pos(1), ...
            'YData', pos(2), ...
            'ZData', pos(3));

        % Trail
```

```
        set(h_trail(i), ...
            'XData', X(1,1:k,i), ...
            'YData', X(2,1:k,i), ...
            'ZData', X(3,1:k,i));

        % Heading arrow (body X-axis)
        R = quat2rotm(quat');
        heading = R(:,1);
        arrow_end = pos + scale_arrow * heading;

        set(h_arrow(i), ...
            'XData', [pos(1) arrow_end(1)], ...
            'YData', [pos(2) arrow_end(2)], ...
            'ZData', [pos(3) arrow_end(3)]);
    end

    title(sprintf('Simulation Time: %.2f s', t(k)));
    drawnow limitrate;

    frame = getframe(gcf);
    writeVideo(v, frame);
end

%% ===== CLOSE VIDEO =====
close(v);
fprintf('Video successfully saved: %s\n', video_name);
```

Listing A.3: M function

```
%This function allow to compiute the matrix M for rotational control

function M = quatM(q)
    q = q(:);
    q0 = q(1);
    qv = q(2:4);

    qx = qv(1); qy = qv(2); qz = qv(3);

    S = [  0    -qz    qy;
          qz     0    -qx;
         -qy    qx     0  ];

    sign_q0 = sign(q0);

    if sign_q0 == 0; sign_q0 = 1; end
    M = [sign_q0 * qv.';
        q0 * eye(3) + S];


end
```

Listing A.4: Target Reference Point

```
%% point_target.m

% Target point
target_point = [10.0; 10.0; 10];  % x, y, z

% desired function and its derivatives
xi_d = @(tt) target_point;
xi_d_dot = @(tt) [0;0;0];
xi_d_ddot = @(tt) [0;0;0];
```

Listing A.5: Cylindrical Trajectory 3D

```
%% trajectory_cylindrical.m


% Parametri cilindro
R = 1.0;            % radius [m]
omega_traj = 0.5; % angular velocity [rad/s]
z0 = 0.0;           % initial altitude
zf = 1.5;           % final altitude
Tend = 10;          % duration
vz = (zf - z0)/Tend;  % vertical velocity

% Desiderd function
xi_d = @(tt) [ R * cos(omega_traj*tt);   % x
               R * sin(omega_traj*tt);   % y
               z0 + vz*tt ];             % z

% I derivate
xi_d_dot = @(tt) [ -R * omega_traj * sin(omega_traj*tt);
                    R * omega_traj * cos(omega_traj*tt);
                    vz];

% II derivate
xi_d_ddot = @(tt) [ -R * omega_traj^2 * cos(omega_traj*tt);
                    -R * omega_traj^2 * sin(omega_traj*tt);
                    0 ];
```

Listing A.6: Spiral Trajectory 3D

```
%% trajectory_spiral.m
% Definition of the desired trajectory for the quadcopter (Conical
    Spiral)

% Spiral Parameters
R_start    = 0.5;         % Initial radius [m]
R_end      = 2.5;         % Final radius [m]
omega_traj = 0.8;         % Angular velocity [rad/s]
z0         = 0.0;         % Initial altitude
zf         = 2.0;         % Final altitude

% Simulation duration
Tend       = 40;

% Growth rates
vz = (zf - z0) / Tend;          % Vertical velocity (constant)
vr = (R_end - R_start) / Tend;  % Radial expansion velocity (constant)

% 1. Desired Position Function
% r(t) = R_start + vr * t
xi_d = @(tt) [ (R_start + vr*tt) * cos(omega_traj*tt);
               (R_start + vr*tt) * sin(omega_traj*tt);
               z0 + vz*tt ];

% 2. Desired Velocity Function
% Uses Product Rule: d/dt(r*cos) = r'*cos + r*(cos)'
xi_d_dot = @(tt) [ vr*cos(omega_traj*tt) - (R_start + vr*tt)*omega_traj
    *sin(omega_traj*tt);
                   vr*sin(omega_traj*tt) + (R_start + vr*tt)*omega_traj
                       *cos(omega_traj*tt);
                   vz ];

% 3. Desired Acceleration Function
% Uses Product Rule again on velocity terms
```

```
% Contains Centripetal term (R*w^2) and Coriolis terms (2*vr*w)
xi_d_ddot = @(tt) [ -2*vr*omega_traj*sin(omega_traj*tt) - (R_start + vr
    *tt)*omega_traj^2*cos(omega_traj*tt);
                     2*vr*omega_traj*cos(omega_traj*tt) - (R_start + vr
                         *tt)*omega_traj^2*sin(omega_traj*tt);
                     0 ];
```

Listing A.7: Lemniscate Trajectory 3D

```
%% trajectory_lemniscate.m

% Parametri lemniscata
A = 1.0;            % Amplitude
omega_traj = 0.5; % angular speed [rad/s]
z0 = 0.0;           % initial altitude
zf = 1.5;           % final altitude
Tend = 10;          % duration
vz = (zf - z0)/Tend;  % vertical velocity

% Lemnicate function
xi_d = @(tt) [ A * cos(omega_traj*tt) ./ (1 + sin(omega_traj*tt).^2);
    % x
               A * sin(omega_traj*tt).*cos(omega_traj*tt) ./ (1 + sin(
                   omega_traj*tt).^2);  % y
               z0 + vz*tt ];  % z

% I derivate
xi_d_dot = @(tt) [ -A*omega_traj*sin(omega_traj*tt)./(1 + sin(
    omega_traj*tt).^2) ...
                    - A*cos(omega_traj*tt).* (2*sin(omega_traj*tt)*
                        omega_traj.*cos(omega_traj*tt)) ./ (1 + sin(
                        omega_traj*tt).^2).^2;

                    A*omega_traj*(cos(omega_traj*tt).^2 - sin(
                        omega_traj*tt).^2) ./ (1 + sin(omega_traj*tt)
                        .^2) ...
                    - A*sin(omega_traj*tt).*cos(omega_traj*tt) .* (2*
                        sin(omega_traj*tt)*omega_traj.*cos(omega_traj*tt
                        )) ./ (1 + sin(omega_traj*tt).^2).^2;

                    vz ];

% II derivate
```

```
xi_d_ddot = @(tt) [ ...
    -A*omega_traj^2 * cos(omega_traj*tt)./(1 + sin(omega_traj*tt).^2)
        ...
    + 4*A*omega_traj^2*sin(omega_traj*tt).^2.*cos(omega_traj*tt).^2./(1
         + sin(omega_traj*tt).^2).^3 ...
    - 2*A*omega_traj^2*sin(omega_traj*tt).*cos(omega_traj*tt).^3./(1 +
        sin(omega_traj*tt).^2).^3;

    -2*A*omega_traj^2*sin(omega_traj*tt).*cos(omega_traj*tt)./(1 + sin(
        omega_traj*tt).^2) ...
    + 2*A*omega_traj^2*(cos(omega_traj*tt).^2 - sin(omega_traj*tt).^2)
        .*sin(omega_traj*tt).*cos(omega_traj*tt)./(1 + sin(omega_traj*tt
        ).^2).^2 ...
    - 4*A*omega_traj^2*sin(omega_traj*tt).^2.*cos(omega_traj*tt).^2./(1
         + sin(omega_traj*tt).^2).^3;

    0 ];
```