Inter IIT Tech Meet 13.0

Robust Control Algorithm for Quadrotor Stability under Single Motor Failure Using NMPC

Endterm Report

Submitted by :
Team 30

Contents

1 Introduction

In this study, we address the problem of detecting and isolating a single motor failure in quadcopters, a critical challenge in UAV operation. A single motor failure can destabilize the quadrotor, risking mission success and safety. This study outlines a N**onlinear Model Predictive Control** (NMPC)-based strategy to ensure stability and safe recovery during motor failures, with motor failure detection using both **machine learning and classical approaches**. By predicting states and optimizing control inputs, the solution addresses the challenge of maintaining control under abnormal conditions.

2 Motor Failure Detection

This study commenced with an extensive literature review on motor failure isolation. To determine the most effective approach, neural network techniques and a classical dynamics method were explored concurrently, allowing for a comparison of their performance in identifying the failed motor.

2.1 Classical Dynamics Approach

Based on the literature reviewed, particularly the works cited in [1], [2], and [3], an observer system was developed to estimate state residuals, with a detailed literature review presented in the midterm submission. These residuals, when analyzed, correspond to different combinations of actuator failures, as illustrated in Figure 2.1.6. Detailed implementation of the observer system are outlined in the following sections.

2.1.1 System Description

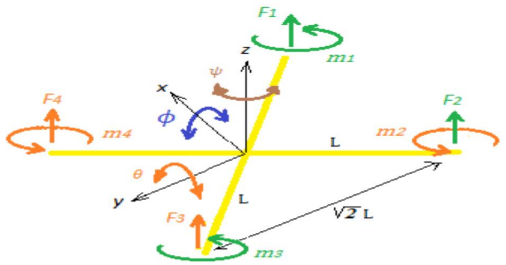


Figure 1: Schematic of the geometric configuration of the quadcopter

The control variables include:

$$\begin{cases} u_1 = \left(\Omega_1^2 + \Omega_2^2 + \Omega_3^2 + \Omega_4^2\right) K_f & \text{throttle control} \\ u_2 = \left(\Omega_3^2 + \Omega_2^2 - \Omega_1^2 - \Omega_4^2\right) LK_f/\sqrt{2} & \text{roll control} \\ u_3 = \left(\Omega_4^2 + \Omega_3^2 - \Omega_2^2 - \Omega_1^2\right) LK_f/\sqrt{2} & \text{pitch control} \\ u_4 = \left(\Omega_1^2 - \Omega_2^2 + \Omega_3^2 - \Omega_4^2\right) K_H & \text{yaw control} \end{cases} \tag{1}$$

Ω_1, Ω_2, Ω_3, and Ω_4 are the speeds of the four motors. And Ω_r is the residual rotational velocity, which is given by:

$$\Omega_r = -\Omega_1 + \Omega_2 - \Omega_3 + \Omega_4.$$

Using the Lagrangian method, the quadrotor rotational dynamic model is as follows :

$$\ddot{\phi} = \frac{\dot{\theta}\dot{\psi}(I_y - I_z) - J_r\dot{\theta}\Omega_r + u_2}{I_x}; \quad \ddot{\theta} = \frac{\dot{\phi}\dot{\psi}(I_z - I_x) + J_r\dot{\phi}\Omega_r + u_3}{I_y}; \quad \ddot{\psi} = \frac{\dot{\phi}\dot{\theta}(I_x - I_y) + u_4}{I_z} \tag{2}$$

where I_x, I_y and I_z represents the moments of inertia along the x, y and z directions, respectively.
By defining the state vector $x^T = \left[\varphi\ \theta\ \psi\ \dot{\varphi}\ \dot{\theta}\ \dot{\psi}\right]$, control input vector $u^T = [u_2\ u_3\ u_4]$, and output vector $y^T = \left[\varphi\ \theta\ \psi\ \dot{\varphi}\ \dot{\theta}\ \dot{\psi}\right]$, the above equations can be described in the state equation as

$$\begin{cases} \dot{x}(t) = Ax(t) + Bu(t) + h(x,u); \quad y(t) = Cx(t) \end{cases} \tag{3}$$

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1/I_x & 0 & 0 \\ 0 & 1/I_y & 0 \\ 0 & 0 & 1/I_z \end{bmatrix}, \quad C = I_{6\times 6}, \quad h(x,u) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{\dot{\theta}\dot{\psi}(I_y - I_z) - J_r\dot{\theta}\Omega_r}{I_x} \\ \frac{\dot{\varphi}\dot{\psi}(I_z - I_x) + J_r\dot{\varphi}\Omega}{I_y} \\ \frac{\dot{\varphi}\dot{\theta}(I_x - I_y)}{I_z} \end{bmatrix}. \tag{4}$$

2.1.2 Setting up the Thau's Observer

A stable observer for the system has the form

$$\begin{cases} \dot{\hat{x}}(t) = A\hat{x}(t) + Bu(t) + h(\hat{x}(t), u(t)) + K(y(t) - \hat{y}(t)); \quad \hat{y}(t) = C\hat{x}(t) \end{cases} \tag{5}$$

where $\boldsymbol{K}$ is the observer gain matrix and is obtained based on Lemma 1.

Lemma 1 If the gain matrix $\boldsymbol{K}$ in Eq. 6 satisfies [1]: $K = P_\theta^{-1} C^T$.

And matrix P_θ is the solution to the Lyapunov equation: $A^T P_\theta + P_\theta A - C^T C + \theta C^T P_\theta = 0$ where θ is a positive parameter which is chosen such that q. 8 has a positive definite solution, then the state of Eq. 6 is an asymptotic estimation of the system state described by Eq. 4, that is, $\lim_{t\to\infty} e(t) = \lim_{t\to\infty}(x(t) - \hat{x}(t)) = 0$ After several simulations, we tuned $\boldsymbol{\theta}$ and found the gain matrix $\boldsymbol{K}$ to be

$$K = \begin{bmatrix} 100 & 0 & 0 & 5 & 0 & 0 \\ 0 & 100 & 0 & 0 & 5 & 0 \\ 0 & 0 & 100 & 0 & 0 & 5 \\ 5 & 0 & 0 & 100 & 0 & 0 \\ 0 & 5 & 0 & 0 & 100 & 0 \\ 0 & 0 & 5 & 0 & 0 & 100 \end{bmatrix} \tag{6}$$

Then, we used Runge-Kutta 4th order (RK4) equations to get $\hat{x}(t)$ from $\dot{\hat{x}}(t)$:

$$\begin{aligned} k_1 &= T_s \cdot \dot{x} \\ k_2 &= T_s \cdot \left(\dot{x} + \tfrac{1}{2} k_1\right) \\ k_3 &= T_s \cdot \left(\dot{x} + \tfrac{1}{2} k_2\right) \\ k_4 &= T_s \cdot \left(\dot{x} + k_3\right) \\ x_{\text{next}} &= x + \tfrac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \end{aligned}$$

2.1.3 Calculating the Residuals

$$\begin{aligned} r_1(t) &= \varphi(t) - \hat{\varphi}(t), \quad & r_3(t) &= \psi(t) - \hat{\psi}(t), \quad & r_5(t) &= \dot{\theta}(t) - \dot{\hat{\theta}}(t), \\ r_2(t) &= \theta(t) - \hat{\theta}(t), \quad & r_4(t) &= \dot{\varphi}(t) - \dot{\hat{\varphi}}(t), \quad & r_6(t) &= \dot{\psi}(t) - \dot{\hat{\psi}}(t). \end{aligned}$$

2.1.4 Integrating the Observer with PX4

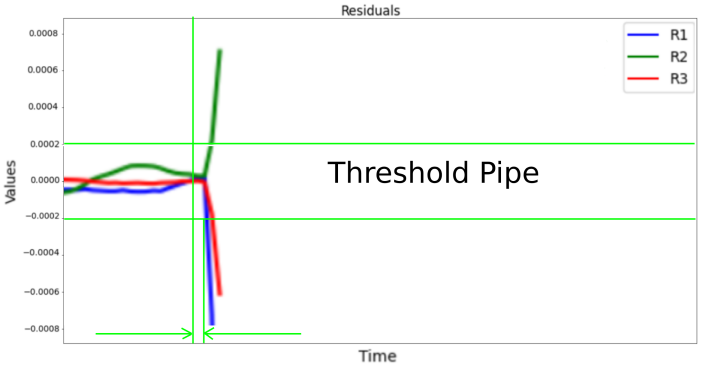
Software-In-The-Loop (SITL) simulation with PX4 was used [4], while interacting within Gazebo Garden's virtual world [5]. Earlier, **eProsima** `uXRCE-DDS` agent was used to locally expose `uORB` topics as `ROS2` [6] topics under namespace `/fmu`. The topics subscribed are mentioned in the table above. The `/motor_failure/motor_number/` topic indicates the failure of

Topic Name	Message Type	Description
`/fmu/out/vehicle_attitude`	`px4_msgs/VehicleAttitude`	Vehicle Orientation
`/motor_failure/motor_number`	`std_msgs/Int32`	Motor failure data
`/fmu/out/actuator_motors`	`px4_msgs/ActuatorMotors`	Motor Thrust Setpoint Data
`/fmu/out/vehicle_angular_velocity`	`px4_msgs/VehicleAngularVelocity`	Vehicle Angular Velocities

Table 1: ROS Topics with Message Types and Data Descriptions

any motor. This is defined by a custom plugin, the specifics of which are provided in the Workspace Setup subsection. For integrating with PX4, a simple application was created within the PX4 codebase as per the official guide [7] that directly subscribed to `uORB` topics, supporting high data streaming rates and reduced latency. All of the Matrix manipulations and multiplications were performed using `Eigen`.

2.1.5 Observations and Fault Estimation



(a) Residuals when Motor 1 is failed

	M_1	M_2	M_3	M_4
r_1	-	-	+	+
r_2	+	+	-	-
r_3	-	+	-	+

Figure 2: Unique map of residuals signs and failed motor

Results:

The model works well for failures occurring during **high-speed linear traversal in any heading** with **100%** accuracy with an average detection latency of **83ms**. Only failure cases are high speed yaw and high speed combined yaw and linear traversal, which are rarely encountered in practice.

2.1.6 Failure Scenarios and Performance

	Low Speed Failure ($2\ \mathrm{m\,s^{-1}}$ *or* $0.2\ \mathrm{rad\,s^{-1}}$)		*High Speed Failure* ($15\ \mathrm{m\,s^{-1}}$ *or* $1\ \mathrm{rad\,s^{-1}}$)	
	Detectability	*Latency* (*ms*)	*Detectability*	*Latency* (*ms*)
Frontal Heading	✓	43.73 ms	✓	140.71 ms
Non − frontal heading	✓	75.78 ms	✓	102.59 ms
Stable Yaw	✓	74.01 ms	✗	-
Traversal + Yaw	✓	60.67 ms	✗	-

2.1.7 Observer gain Matrix calculation for the nonlinear systems

The nonlinear systems described by: $\dot{x}(t) = Ax(t) + Bu(t) + h(x,u); \quad y(t) = Cx(t).$ The functions $h(x,u)$, is a Lipschitz nonlinearity with a Lipschitz constant γ, i.e. $\|h(x,u) - h(\hat{x},u)\| \leq \gamma\|x - \hat{x}\|, \quad \forall x, \hat{x}$
The observer is given by:

$$\begin{cases} \dot{\hat{x}}(t) = A\hat{x}(t) + Bu(t) + h(\hat{x}(t),u(t)) + K(y(t) - \hat{y}(t)); \quad \hat{y}(t) = C\hat{x}(t) \end{cases} \tag{7}$$

Theorem: For the class of systems and observer forms described in equations (7)-(8) and (9), if an observer gain matrix **K** can be chosen such that

$$\begin{bmatrix} (A-KC)^T P + P(A-KC) + \varepsilon\gamma^2 I & P \\ P & -\varepsilon I \end{bmatrix} < 0 \tag{8}$$

for some positive definite symmetric matrix P, then this choice of K leads to asymptotically stable estimates by the observer (9) for the system (7). **Equation (10) is both a necessary and sufficient condition for observer stability.**
Less Conservative Lipschitz Condition
It is possible to write equation (8) in the matrix form defined as

$$\|h(x,u) - h(\hat{x},u)\| \leq \|G(x - \hat{x})\|, \quad \forall x, \hat{x} \tag{9}$$

Note that the matrix G in this case could be a sparsely populated matrix. Hence $\|G(x-\hat{x})\|$ can be much smaller than the constant $\gamma\|x-\hat{x}\|$ used earlier in equation (8) for the same nonlinear function.
Corollary to Theorem 1 For the class of systems and observer forms described in equations (7)-(8) and (11), if an observer gain matrix **K** can be chosen such that

$$\begin{bmatrix} (A-KC)^T P + P(A-KC) + \varepsilon G^T G & P \\ P & -\varepsilon I \end{bmatrix} < 0 \tag{10}$$

for some positive definite symmetric matrix P, then this choice of K leads to asymptotically stable estimates by the observer (9) for the system (7).

Reformulation of Observer Design Using Riccati Equations
Equations (10) and (12) are LMIs involving two unknown matrices K and P . The LMIs can be replaced by a Riccati inequality in just one variable P. A necessary and sufficient condition in term of the existence of a solution to a Riccati inequality can then be obtained. The following theorem summarizes the result.
Theorem: There exists an observer of the type given by equation (9) for the system given by equations (7) and (11) such that the error dynamics are quadratically stabilized if and only if there exist $\varepsilon > 0$ and $\beta \in \mathbb{R}$ such that the following Riccati inequality has a symmetric positive definite solution P: $A^T P + PA + \varepsilon G^T G + \frac{1}{\varepsilon} PP - \beta^2 C^T C < 0$
The observer gain can then be chosen as $K = \frac{\beta^2}{2} P^{-1} C^T$. P can be found using MATLAB command "ARE":

$$P = ARE(a,b,c)$$

where $a = -A$, $b = \frac{1}{\varepsilon} I$ and $c = -\varepsilon G^T G + \beta^2 C^T C - \mu I$ (where $\mu > 0$, μ is a small scalar)

Caution: Gain Selection and Sensitivity Trade-off

A higher value of the gain matrix **K** reduces the detection time but increases the observer's sensitivity to noise. Therefore, selecting an optimal value for **K** is crucial. During agile flights, the use of a high gain led to false detections, highlighting the trade-off between detection speed and noise robustness.

2.2 Machine Learning based Approach

Several studies on motor failure detection in UAVs, based on machine learning were reviewed. Sadhu et al. in [8] proposed a deep learning approach combining CNN and BiLSTM-based autoencoders for anomaly detection, achieving 95% accuracy after five windows, but with latency unsuitable for real-time detection. Zhang et al. in [9] used vibration signal analysis for fault detection in propellors, but the method introduced significant latency when adapted for real-time motor failure detection. These approaches are described in detail in the midterm solution.

2.2.1 Our Approach

After reviewing various approaches, a **lightweight Recurrent Neural Network (RNN)** model was implemented for motor failure detection. The model, shown in Figure 3, consists of an input layer, two hidden layers, and an output layer, with predictions made by **applying a threshold** to the output values.

Note

The model is designed to be very efficient, **with only 31,000** trainable parameters, resulting in a compact size of 0.13 MB.

RNNs offer several advantages in this particular example, including:

- RNNs are suited for sequential data processing because they can model temporal dependencies in sequential data by retaining information from previous time steps.
- By vanishing older data in the forward pass, the RNN naturally focuses on the most recent and relevant sensor data for accurate detection.

The model is trained on **10 IMU-derived features:** $\dot{\theta}_x$, $\dot{\theta}_y$, $\dot{\theta}_z$ (angular accelerations), a_x, a_y, a_z (linear accelerations), q_x, q_y, q_z, q_w (orientation).Each input has shape $(1, 10)$ and the model predicts the failed motor based on these features.

2.2.2 Data Collection

The training dataset includes the 10 sequential features and an additional column indicating the failed motor at each timestamp, serving as the target label. Data was collected as described in Section 2.1.4.To train the model, 50 flights of 30s duration each were recorded for the training dataset and 20 flights for testing purposes. The collected data was converted into a CSV file for structured input.

Few Points to Note:

- Data collection in the Z-direction stopped if the drone's altitude dropped below 0.5 meters to avoid recording erroneous crash data, which could mislead the model by linking crashes to motor failure.
- To prevent the network from inferring a topology among motor indices, the model was trained to predict a one-hot vector, where the position of 1 indicates the failed motor's index.

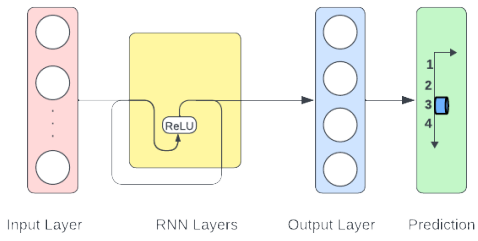


Figure 3: Model Layers

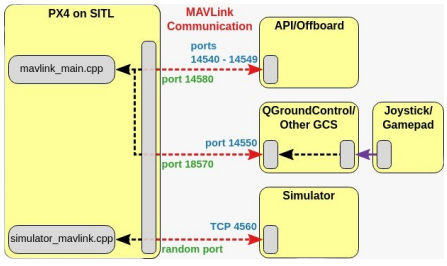


Figure 4: SITL Simulation

2.2.3 Training and Testing

The input for the model consists of rows from CSV files, each containing an input vector of shape $(1, 10)$, derived from IMU sensor features. Each vector is paired with a corresponding label, representing the failed motor at that timestamp. The model processes this sequential data, leveraging its temporal memory to detect motor failures in real-time with high accuracy. Figure 16 shown in the appendix, depicts the **reduction of mean squared loss** during training.

2.2.4 Reliability-Latency Trade-Off

The threshold value significantly affects model performance, balancing reliability and latency in motor failure detection. A low threshold (e.g., 0.3) enables faster detection but may lead to false positives, compromising reliability. A high threshold (e.g., 0.95) improves reliability by reducing false positives but increases latency, delaying failure detection.

Figures 5 and 6 illustrate this trade-off. Figure 5 shows a decrease in the number of columns with predictions above the threshold as it increases, indicating higher precision. Figure 6 demonstrates that higher thresholds result in increased latency, as the time to exceed the threshold after the first true value grows.

After evaluating this trade-off, a threshold of 0.8 was chosen to effectively balance reliability and latency, ensuring accurate detection with zero false positives. As shown in Figure 15 in the appendix, the average latency is 9 steps (180ms at 150Hz), confirming the threshold's effectiveness in real-world conditions.

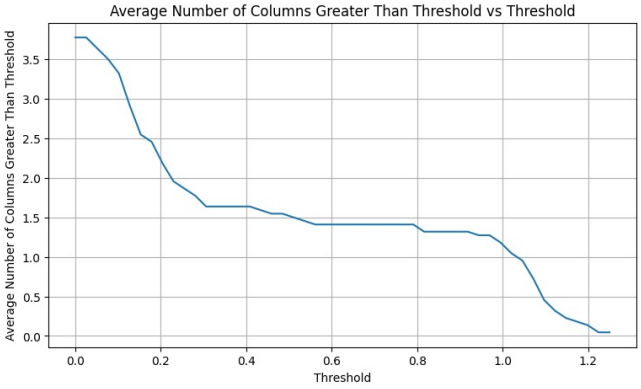


Figure 5: Number of Columns Greater than Threshold vs Threshold

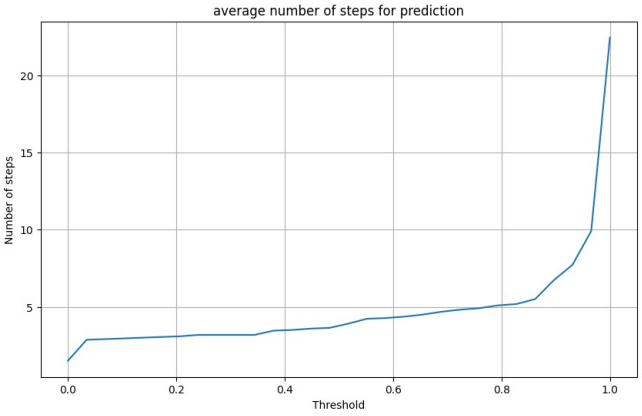


Figure 6: Number of Steps vs Threshold

The confusion matrix at a 0.8 threshold (Figure 7) shows strong diagonal values, indicating good classification of motor failures with minimal misclassifications. This supports the choice of a 0.8 threshold.

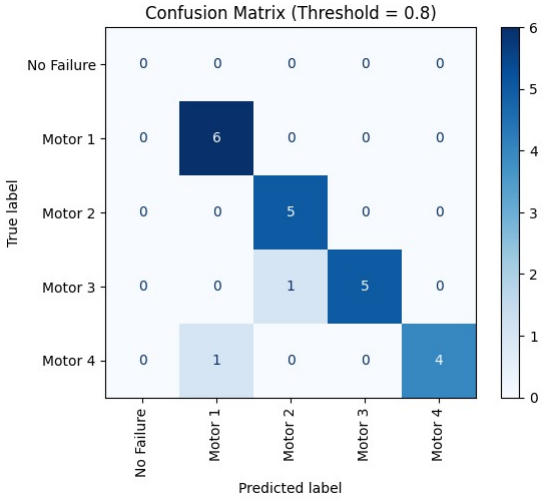


Figure 7: Confusion Matrix at Threshold = 0.8

3 Recovery and Post-Failure Control

Upon the detection of a motor failure, the reference state of the quadrotor is captured and stored immediately before the failure. This reference state is used as a **target for recovery**, allowing the quadrotor to return to its prior position. The algorithm then works to stabilize the quadrotor, achieving a controlled hovering state at this captured position. By prioritizing a return to the pre-failure reference point, deviation is minimized and safety is ensured in response to an unexpected motor failure. Literary references are heavily studied, with a summary provided below.

- **Sun et al.** [10] investigate fault recovery for quadrotors subjected to complete rotor failures under significant disturbances. They employ a combination of **Incremental Nonlinear Dynamic Inversion (INDI)** and **Non-Singular Terminal Sliding Mode Control (NTSMC)**. INDI provides adaptive compensation by incrementally adjusting control inputs, ensuring stability over time, whereas NTSMC utilizes terminal sliding surfaces to guide the system state to a desired target within a finite period. This dual approach effectively manages large faults, though INDI may lack precision for minor disturbances, and NTSMC might overcompensate for smaller faults.
- **Lippiello et al.** [11] present a **PID-based approach** for emergency landings in quadrotors experiencing propeller failures. Their method involves adjusting the remaining motors using a PID controller to maintain a stable hover and execute an emergency landing. While effective in stabilizing the quadrotor, the loss of thrust from a failed motor

significantly reduces the drone's lifting capacity, limiting its ability to clear obstacles and reach the intended landing zone.

- **Fang Nan** [12] integrates a **Nonlinear Model Predictive Controller (NMPC)** with a custom-defined cost function to achieve robust stabilization, recovery, and controlled landing of quadrotors post-motor failure. NMPC accounts for both the **nonlinear dynamics and specific thrust limitations** of each motor, providing precise and reliable responses in failure scenarios. Implementation demonstrated stability under extreme conditions, enabling the quadrotor to return to the pre-failure reference position, achieve a stable hover, and perform a controlled descent for a safe landing.

3.1 Core Principles and Logic

3.1.1 State Variables

The state variables are the characteristics of the quadrotor which can effectively describe all the properties of the quadcopter at any point of time and using them the motion of the quadrotor can be predicted. Our model considers the following state variables :

Category	Notation	Description
Position	$\mathbf{p}(x, y, z)$	Position of the drone in the global coordinate system.
Orientation	$\mathbf{q}(q_1, q_2, q_3, q_4)$	Orientation of the drone as a quaternion.
Velocity	$\mathbf{v}(v_x, v_y, v_z)$	Velocity of the drone along the x, y, and z axes.
Angular Velocity	$\mathbf{w}(w_x, w_y, w_z)$	Angular velocity of the drone along roll, pitch, and yaw axes.
Thrust	$\mathbf{t}(t_1, t_2, t_3, t_4)$	Thrust produced by each motor of the drone.
Control Input	$\mathbf{u}(u_1, u_2, u_3, u_4)$	Commanded thrust for each motor.

Table 2: Drone State and Control Variables

3.1.2 System Dynamics

The dynamics of a quadrotor with motor failure can be modeled using quaternion representation for orientation and incorporating motor delay dynamics. Let $\mathbf{p}$ denote the position vector and $\mathbf{v}$ the velocity of the quadrotor's center of gravity in the inertial frame. Let $\mathbf{q} = [q_w, q_x, q_y, q_z]^T$ represent the quaternion orientation, and $\boldsymbol{\omega} = [\omega_x, \omega_y, \omega_z]^T$ represent the angular velocity in the body frame. The dynamic model is given by:

$$\dot{\mathbf{p}} = \mathbf{v}, \tag{11}$$

$$\dot{\mathbf{q}} = \frac{1}{2}\mathbf{q} \circ \begin{bmatrix} 0 \\ \boldsymbol{\omega} \end{bmatrix}, \tag{12}$$

$$\dot{\mathbf{v}} = \mathbf{q} \otimes \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix} - \mathbf{g}, \tag{13}$$

$$\dot{\boldsymbol{\omega}} = \mathbf{I}^{-1} \left(\boldsymbol{\tau} - \boldsymbol{\omega} \times (\mathbf{I}\boldsymbol{\omega}) + \boldsymbol{\tau}_{\text{ext}} \right), \tag{14}$$

Where:

- T is the total thrust,
- $\boldsymbol{\tau}$ is the total torque generated by the actuators,
- $\mathbf{I}$ is the inertia matrix,
- $\circ$ denotes quaternion multiplication, and
- $\boldsymbol{\tau}_{\text{ext}}$ represents unmodeled external torques, including aerodynamic disturbances.

3.1.3 Cost Function and Constraints

Define the state vector as $\mathbf{x} = [\mathbf{p}^T, \mathbf{q}^T, \mathbf{v}^T, \boldsymbol{\omega}^T, \mathbf{t}^T]^T$ and control input vector as $\mathbf{u} = [u_1, u_2, u_3, u_4]^T$. The NMPC objective is to minimize:

$$\min_{\mathbf{u}_{k:k+N-1}} \mathbf{y}_N^T \mathbf{Q}_N \mathbf{y}_N + \sum_{i=k}^{k+N-1} \left(\mathbf{y}_i^T \mathbf{Q} \mathbf{y}_i + \mathbf{u}_i^T \mathbf{R} \mathbf{u}_i \right)$$

The cost function consists of 2 parts: **the terminal cost** and **the staging cost**.

- In our NMPC approach, a **staging cost**, or stage cost, is applied at each step within the prediction horizon. This incremental cost functions as a guiding metric, continuously penalizing any deviations in both state and control inputs throughout the process. This way, the controller actively works to keep the system on track at every point in the horizon.
- The **terminal cost** serves as a focused constraint applied exclusively at the final step of the prediction horizon. This cost penalizes deviations in the state at the horizon's endpoint, providing a strong incentive for the system to converge closely to a desired final state. This acts as an endpoint anchor, guiding the system toward stability as it approaches the final predicted state.

This is not sufficient so we also subject the cost function to a few constraints :

- The state of the quadrotor are controlled by the following:

$$\mathbf{x}_{i+1} = f(\mathbf{x}_i, \mathbf{u}_i), \quad i = k, \ldots, k+N-1 \tag{15}$$

This relationship emerges directly from the **nonlinear dynamics** we defined for our quadrotor. By avoiding any linearization within our model, we retain the full complexity of the drone's behavior. As a result, our model is more accurate and responsive to real-world conditions, enabling a more effective control strategy without sacrificing precision or robustness.

- We apply these limits to input we shall provide to each of the motors:

$$\mathbf{u}_{\min} \leq \mathbf{u} \leq \mathbf{u}_{\max}, \tag{16}$$

This approach mirrors real-world conditions, as each motor has a **physical limit on the maximum thrust** it can produce. By incorporating this constraint, our solution ensures that the control outputs remain realistic and achievable within the motors' capabilities.

The weight matrices $\mathbf{Q}$, $\mathbf{Q}_N$, and $\mathbf{R}$ are positive-definite and were optimally obtained as follows:

$$\mathbf{Q} = \text{diag}(80,\ 80,\ 80,\ 3,\ 0,\ 0.5,\ 0.5,\ 0.5,\ 5,\ 5,\ 0.4)$$
$$\mathbf{Q}_N = \text{diag}(80,\ 80,\ 80,\ 3,\ 0,\ 0.5,\ 0.5,\ 0.5,\ 5,\ 5,\ 0.4)$$
$$\mathbf{R} = \text{diag}(0.1,\ 0.1,\ 0.1,\ 0.1)$$

Upon detecting a motor failure, we can simulate the failure by setting the control input constraint for the affected motor. Specifically, we **adjust the minimum and maximum allowable values for the motor's thrust to zero**, effectively rendering it inactive.

3.1.4 Attitude Error and Cost Vector

We compute the error which and the cost vector i$\mathbf{y}_i$ in this subsection.

- **Quaternion Error :** For the error, all the quantities except the quaternions could be simply subtracted from each other as they are always achievable. However since we can never control the attitude of the drone completely with 3 motors, there arises a need for an alternate way of computing its error. First we assume a $\mathbf{q}_{\text{ref}}$ to obtain. This would be [1, 0, 0, 0] as it corresponds to a stable position with horizontal in the x-y plane. To compute the attitude error we do.

$$\boxed{\mathbf{q_e} = \mathbf{q_r ef} \circ \mathbf{q}^{-1}} \tag{17}$$

Once we have this we perform an split in the quaternion error. We divide it into the $\mathbf{q}_{\text{xy}}$ and $\mathbf{q}_{\text{z}}$.

> **Note**
>
> This is because $\mathbf{q}_{\text{z}}$ can never be controlled properly with 3 motors, so we must separate the controllable part of the quaternion. Here $\mathbf{q}_{\text{z}}$ is obtained by isolating the scalar and the z component of the quaternion error we obtained in the earlier step.

To compute $\mathbf{q}_{\text{xy}}$ term, we perform the following operation.

$$\boxed{\mathbf{q_{xy}} = \mathbf{q_e} \circ \mathbf{q_z}^{-1}} \tag{18}$$

- **Angular Velocity Error :** Another challenge we faced was defining a realistic reference angular velocity, $\mathbf{w}_{\text{ref}}$, for the system.

> **Note**
>
> Setting $\mathbf{w}_{\text{ref}} = \mathbf{0}$ is not feasible with only three functioning motors, as achieving zero angular velocity across all axes is unattainable under these conditions. Consequently, we needed to identify an optimal, $\mathbf{w}_{\text{ref,z}}$ stable that would support controlled hover while accommodating the system's limitations.

To determine this reference, we analyzed the stable hover angular velocity across different altitudes, testing values that allowed the drone to maintain stability despite the motor constraint. We identified that the best results were obtained at an altitude of (0,0,5), with an optimal $\mathbf{w}_{\text{ref}}$ value of approximately 21.348. This angular velocity provided a balanced and achievable target that enabled the drone to sustain a steady hover, effectively compensating for loss of one motor.

- **Position Error :** The position references are straightforward to define, as they simply correspond to the **target positions** the drone needs to reach. These targets serve as the goal for the drone's movement and guide its setpoints.
- **Other State Errors :** For the other state references, such as velocities and accelerations, we set them to zero. This ensures that the drone is not attempting to move away from the target position, and it encourages **stability at the desired location**.

Establishing all these references, the cost vector used in our cost function can be expressed as:

$$\mathbf{y}_i = \begin{bmatrix} \mathbf{p} - \mathbf{p}_{\text{ref}} \\ q_{xy,x}^2 + q_{xy,y}^2 \\ q_z^2 \\ \mathbf{v} - \mathbf{v}_{\text{ref}} \\ \omega - \omega_{\text{ref}} \\ \mathbf{t} - \mathbf{t}_{\text{ref}} \\ \mathbf{u} - \mathbf{u}_{\text{ref}} \end{bmatrix}$$

where q_{xy} and q_z represent the pitch-roll and yaw errors, respectively.

3.1.5 Implementation of NMPC using CasADi

Initially, we attempted to implement the **NMPC** for quadrotor recovery entirely using **CasADi**. CasADi's symbolic framework allowed us to easily define the quadrotor dynamics, including failure scenarios, and to formulate the nonlinear optimization problem required for NMPC. However, despite the flexibility and ease of use CasADi offers, we encountered significant **computational latency** when attempting to solve the NMPC optimization problem in real time.

Note

In practice, the CasADi-based NMPC framework introduced a delay of approximately **2 seconds** in generating the optimized control inputs. This latency rendered the controller unsuitable for real-time applications, as the quadrotor's response to motor failure could not be adjusted promptly.

The delay was mainly due to the fact that CasADi, while efficient for symbolic model setup and derivative calculations, is **not optimized as a standalone solution** for rapid execution of complex nonlinear optimization tasks in real time.

To address this issue, we incorporated **ACADOS as the NMPC solver**, integrating it with CasADi for symbolic modeling while using ACADOS to handle the real-time optimization. ACADOS is specifically designed for fast, efficient solutions to **nonlinear optimal control problems (OCPs)** and is optimized for applications requiring high-speed performance.

Note

Upon switching to ACADOS, the 2-second latency vanished, and the NMPC was able to generate control inputs in real time. This improvement allowed the quadrotor to maintain stability and trajectory tracking immediately after a motor failure was detected.

The successful integration of CasADi and ACADOS combined ease of model definition with the computational efficiency required for real-time control. This hybrid approach enabled us to achieve fault-tolerant control for the quadrotor, meeting the performance standards needed for effective NMPC in real-time flight scenarios.

3.1.6 ACADOS for Real-Time NMPC Optimization

To address the problem, we initially employed a predefined cost function in ACADOS called `'LINEAR_LS'`. This cost function takes the reference values, subtracts the actual state values, and squares the results. By minimizing this squared error, the cost function drives the solution toward an optimal set of control actions that reduces the gap between the reference trajectory and the actual system trajectory.

The results obtained from this method demonstrated the system's ability to **track the goal with moderate success**, but its performance in trajectory tracking was significantly lacking. While the optimization of the cost function helped reduce some deviations, the system **struggled to maintain stability in the control process**, leading to suboptimal outcomes in trajectory tracking and overall poor performance in our optimization task.

Subsequently, we adopted an external cost function, as outlined in Section 4.4, referred to as the `'EXTERNAL'` cost function. This transition presented significant challenges, as it required an overhaul of the solver configurations and weight parameters that were initially tailored for the `'LINEAR_LS'` cost function.

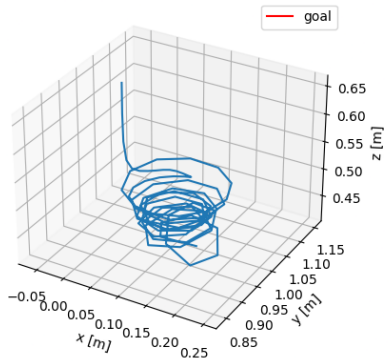
Note

The `'EXTERNAL'` cost function offered increased flexibility and customization potential compared to `'LINEAR_LS'`, allowing us to provide a custom cost function. However, implementing it involved re-evaluating each parameter to ensure alignment with the new cost function.

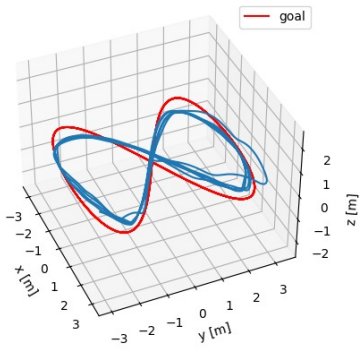
Below, we outline the key changes made to the solvers, detailing the adjustments necessary to achieve compatibility with the EXTERNAL cost function and ensure that the optimization could proceed smoothly. These changes were crucial in adapting our approach to maintain system performance and stability under the revised cost criteria.

- **QP Solver :** We chose the `'PARTIAL_CONDENSING_HPIPM'` solver for the `QP` problem at each timestep. This solver is specifically designed to handle large-scale optimization problems efficiently by leveraging the `High-Performance Interior-Point Method (HPIPM)` with partial condensing.
- **Integrator :** For the system's dynamics, we used the `'ERK'` (Explicit Runge-Kutta) integrator. The `ERK` integrator is known for its efficiency and accuracy when dealing with continuous nonlinear systems, making it a natural choice for the quadrotor model.
- **NLP Solver :** We configured the solver to use the `SQP_RTI` (Sequential Quadratic Programming with Real-Time Iteration) solver type. The `Real-Time Iteration (RTI)` aspect ensures that updates to the solution are incremental, which allows for faster convergence at every timestep. This iterative approach is key for achieving real-time performance in fast-moving systems such as a quadrotor.
- **Approximator :** Additionally, we selected the `Gauss-Newton` method for `Hessian` approximation. Calculating the exact Hessian matrix can be computationally expensive, so the this was used to reduce the complexity of the optimization problem.

These solver settings combined to form an efficient NMPC controller capable of real-time flight control, even under fault conditions such as motor failure.



(a) Trajectory tracking using Linear_LS Type Cost Function



(b) Trajectory Tracking using EXTERNAL Type cost function

3.1.7 INDI for mitigating external disturbances

During simulation on Gazebo, the effects of air drag and sensor data noise were observed to affect the controller's performance. Torque is recognized as the most sensitive parameter in the dynamics of the UAV, and accurately modeling external forces such as drag is difficult due to the complexity of Gazebo dynamics. To address this issue, INDI (Incremental Nonlinear Dynamic Inversion) has been used to approximate the external thrust caused by drag, which is challenging to model precisely within the Gazebo dynamics.

How INDI helps?

The INDI approach relies on sensor readings to calculate the actual torque on the UAV, which is then compared with the theoretical torque calculated using a dynamic model. The difference between these torques is taken as the external torque, which is used to refine the thrust command. This helps mitigate the effects of external forces such as drag, improving the overall thrust values and performance of the UAV.

The external torques are estimated as follows:

$$\boxed{\tau_{\text{ext}} = I_v\dot{\omega}_f - \tau_f + \omega_f \times I_v\omega_f} \tag{19}$$

Where the subscript f indicates that the variable is measured and low-pass filtered for noise reduction. τ_f is the estimated torque obtained from the model, with the force of each rotor t being determined from low-pass filtered rotor speed measurements.

By substituting this external torque into the system dynamics, the following corrected equations for the thrust and angular acceleration are derived:

$$\boxed{\mathrm{I}_v\dot{\omega} = I_v\dot{\omega}_f + \tau - \tau_f + (\omega_f \times I_v\omega_f - \omega \times I_v\omega) \approx I_v\dot{\omega}_f + \tau - \tau_f} \tag{20}$$

From the NMPC, the thrust command u is used to obtain the desired collective thrust T_d and angular acceleration α_d:

$$\begin{bmatrix} T_d \\ I_v\alpha_d + \omega \times I_v\omega \end{bmatrix} = Gu \tag{21}$$

The desired torque is then calculated using these values:

$$\boxed{\tau_d = \tau_f + I_v(\alpha_d - \dot{\omega}_f)} \tag{11}$$

Finally, the thrust command for each rotor is calculated using the reduced control effectiveness matrix $\hat{G}$:

Where G is the normal control matrix, with the i-th column values set to zero to account for the failure of the i-th rotor. This corresponds to the rotor that is not functioning, ensuring that the thrust command for the failed rotor is zero.

$$u_{\text{indi}} = \hat{G}^{+} \begin{bmatrix} T_d \\ \tau_d \end{bmatrix} \tag{22}$$

This approach has been used to improve the thrust command accuracy by adjusting for external torques caused by drag, which are difficult to model directly. The use of INDI ensures better adaptation of the control inputs, mitigating the effects of disturbances such as drag.small ranges

3.2 Simulation and Testing

3.2.1 Preliminary Testing

The CasADi-ACADOS based NMPC was initially tested in dynamics simulation implemented in python for various trajectories, with one motor failing at predefined instances. The results demonstrated the effectiveness of the NMPC controller in stabilizing the quadrotor under motor failure conditions, showing promising control capabilities for fault-tolerant flight.

Gazebo Testing

Post preliminary testing, the controller was run after integration with PX4 with Gazebo simulation, the graphs depicting these are attached in the appendix.They depict motor failure occuring at the 200th time step and the subsequent recovery of the drone. These results have been provided in the simulation videos submitted. The changes made to the simulation setup have been elaborated in subsequent sections.

- **Stable Hover**

After motor failure, for attaining stable hover thrust have been adjusted in the remaining three motors such that after reaching a goal point it hovers at that point. This is simulated by the StableHover() function in the code. The results are shown in Figure 9. For more details refer to the README.md file in the codebase submitted.

- **Controlled Landing** After achieving a stable hover, our drone can create a helical path directly beneath it, gradually reducing its speed as it approaches the landing position. This is simulated by the ControlledLanding() function in the code. The graphs in Figure 10 depict landing with an initial position = [0, 0, 10] initial velocity = [1, 3, 2], angular velocity = [1, 2, 2] and motor thrusts = [2, 2, 3, 4]

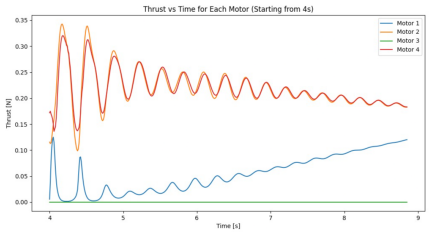
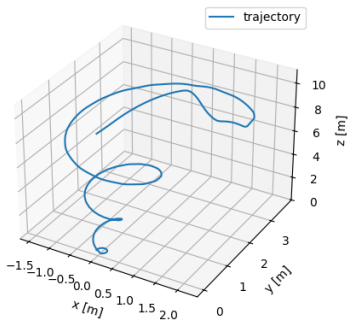
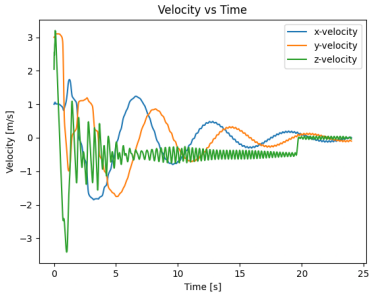


Figure 9: Thrust for each motor while stable hover, post motor failure



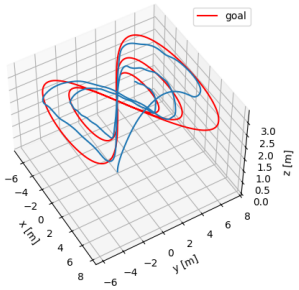
(a) Trajectory tracked during emergency landing



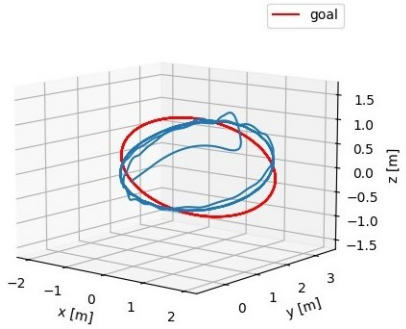
(b) Velocity in x, y, z during controlled landing

Figure 10: Depiction of Controlled Landing

- **Basic Navigation** The following graphs show the drone's navigation performance following a given trajectory, simulated by the BasicNavigation() function. For further details, refer to the README.md file provided in the documentation.



(a) Decaying radius 8 with initial z=3



(b) Ellipse in the xz plane with major axis along x and minor along z

3.2.2 ROS Integration with PX4

The goal of this task was to switch the PX4 flight controller to `OFFBOARD` mode, enabling autonomous control of the drone. Two main tasks were involved:

1. **Publishing the `OffboardControlMode` message** at a consistent rate.
2. **Sending a `VehicleCommand` message** to trigger the mode switch.

Entering OFFBOARD Mode To enter `OFFBOARD` mode, the `OffboardControlMode` message was published at a frequency of 10 Hz on the `/fmu/in/offboard_control_mode` topic. This maintained the required heartbeat signal for PX4. All parameters of the `OffboardControlMode` message were set to `False`, except for the `direct_actuators` parameter, which was set to `True`. This configuration informed the controller that the drone was to be controlled by direct actuator commands. The function `publish_offboard_control_heartbeat_signal()` continuously published this message.

Activating OFFBOARD Mode Once the `OffboardControlMode` message was being published regularly, the `OFFBOARD` mode was activated as follows:

1. A `VehicleCommand` message was sent to the `/fmu/in/vehicle_command` topic just before arming the drone.
2. The `command` parameter in the message was set to `VehicleCommand.VEHICLE_CMD_DO_SET_MODE`.

The drone was armed immediately after sending the mode switch message by sending the same `VehicleCommand` from `PX4_msgs.msg` with the following parameters:

- `command = VehicleCommand.VEHICLE_CMD_COMPONENT_ARM_DISARM`
- `param1 = 1`

This sequence was executed in the following functions:

```
self.engage_offboard_mode()
self.arm()
```

Functions:

- `engage_offboard_mode()`: This function switches the drone to `OFFBOARD` mode by publishing the `VehicleCommand` message with the appropriate parameters.
- `arm()`: This function arms the drone by publishing a `VehicleCommand` message to activate the motors.

NMPC Controller State Retrieval For the Nonlinear Model Predictive Control (NMPC) controller to function, it required the current state of the drone, which was retrieved from the `VehicleOdometry` message published by PX4 on the `/fmu/out/vehicle_odometry` topic. This message contained the following parameters:

Parameter Type	Components
Position	`x`, `y`, `z`
Orientation	`q1`, `q2`, `q3`, `q4` (quaternion)
Linear Velocities	`vx`, `vy`, `vz`
Angular Velocities	`roll`, `pitch`, `yaw` rates

Table 3: Parameters Description for VehicleOdometry

Main Control Loop The `main.py` file handled communication with PX4 by retrieving the drone's state and passing it to the `run_optimizer()` function of the controller. The `run_optimizer()` function computed the normalized thrust values for the motors, which were then published as an `ActuatorMotors` message to PX4 on the `/fmu/in/actuator_motors` topic.

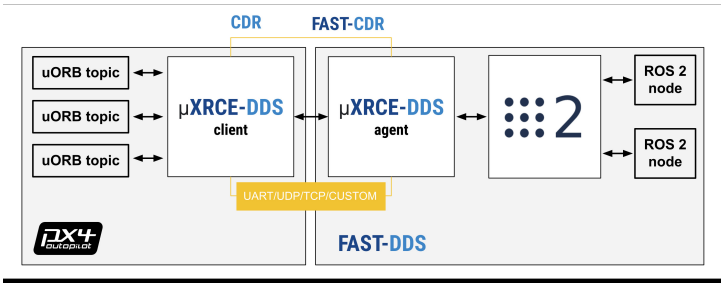


Figure 12: ROS Integration with PX4 for Autonomous Drone Control

The image demonstrates the setup for integrating the PX4 flight controller with ROS to enable autonomous control by switching to OFFBOARD mode. The integration involved two main steps: publishing the `OffboardControlMode` message at a consistent rate to maintain a heartbeat signal, and sending a `VehicleCommand` message to trigger the mode switch. This setup ensures that the drone can be controlled directly through actuator commands, as indicated by the *direct_actuators* flag in the message, thus allowing autonomous flight control in the ROS environment

3.2.3 Modifications Made:

This subsection summarizes the modifications made to the Gazebo simulator during controller testing. Motor failure detection was performed using Gazebo Classic, while the controller operates within Gazebo Garden. Challenges encountered in integrating the controller with PX4 and Gazebo are discussed later

Gazebo Garden Advantages:

- **Modernized Architecture and Advanced Features**: Gazebo Garden's modular design based on Ignition Robotics enhances scalability and maintainability. Its modern API supports multi-robot simulations, sophisticated physics engines, and detailed sensor modeling, enabling advanced simulation capabilities.
- **Optimized Performance**: Improved multi-threading, rendering, and resource utilization allow Gazebo Garden to efficiently handle large and complex simulations.
- **Seamless ROS 2 Integration and Accessibility**: With native ROS 2 support, Gazebo Garden ensures smooth communication and development. Its user-friendly interface, active ecosystem, and comprehensive documentation enhance accessibility for developers and researchers.

Modifications to `model.sdf` File

1. **Modification to Motor Constant:**
   In the `PX4-Autopilot/Tools/simulation/gz/models/x500/model.sdf` file, update the motor constant to ensure adequate thrust for stable flight. See the changes below:

Original Value	Modified Value
`<motorConstant>8.54858e-06</motorConstant>`	`<motorConstant>20e-06</motorConstant>`

2. **Modification to Time Constant:**
   In the same file, update the time constants to remove delays in attaining desired thrust. See the changes below:

Original Values	Modified Values
`<timeConstantDown>0.025</timeConstantDown>`	`<timeConstantDown>0</timeConstantDown>`
`<timeConstantUp>0.0125</timeConstantUp>`	`<timeConstantUp>0</timeConstantUp>`

Difference from Gazebo Classic Setup: In contrast to the Gazebo Classic setup (`make px4_sitl gazebo-classic`), the Gazebo Garden configuration employs `gz_x500` and modifies specific simulation parameters to ensure compatibility with the updated Gazebo environment.

The testing process included:

- Fine-tuning with four motors for stable hover
- Transitioning to stable trajectory following
- Testing motor failure with a three-motor controller triggering motor failure

4 Challenges Faced and Lessons Learned

The integration process of the controller with the gazebo simulation presented significant challenges and proved to be a humbling experience.

4.1 Challenges in Failure Detection

4.1.1 Classical Observer Detection Method

- **Coupled Motion:** Highly coupled motion where Roll, Pitch and Yaw change simultaneously at very high rates, are hard to monitor for failure detection and result in low-accuracy detections.
- **Frame of Residuals:** Observers used in existing literature calculate residuals with respect to **PLUS** dynamics. Using the residuals calculated in **CROSS** dynamics to estimate fault is meaningless. Since rotations are linear transformations, we can directly 'Rotate' these residuals along an up-facing axis by **45-degrees**

4.1.2 Machine Learning Method

- **Training Data Acquisition**: Training data was exclusively obtained through simulation, as testing with actual hardware required infeasible drone crashes, resulting in model errors upon deployment.
- **Erroneous Predictions in Simulation**: During PX4 simulation, erroneous predictions were generated by the model during the takeoff phase, with the cause remaining unidentified despite correct performance when data was processed remotely.
- **Manual Dataset Creation Challenges**: Manual dataset creation was impeded by the necessity to provide waypoints each time, making the process time-consuming and limiting the dataset's size and diversity.

4.2 Challenges in Post Failure Recovery Simulation

4.2.1 Difference in Dynamic Modelling

Testing showed that PX4 models its dynamics differently from our controller, causing axis misalignment: PX4 inverts the Z-axis and rotates the X and Y axes by 90 degrees (see Figure 13). To resolve this, we rewrote the dynamics in PX4's frame of reference and mapped our controller's rotors to PX4. This mapping involved experimentally determining PX4's motor numbering by sequentially applying thrust to each motor and observing its movement, then aligning it with the controller's motor numbering.

4.2.2 Switching between two Controllers

Initially, **motor failure** was handled by switching between two controllers: one for normal operation with no constraints and another for failure mode, where one motor was constrained to zero thrust. However, this approach lacked adaptability in dynamic failure scenarios.

To improve, we integrated **dynamic constraint adjustments** using the **ACADOS solver**. Instead of using separate controllers, motor constraints were treated as parameters within the solver. During flight, the thrust bounds of functional motors remain flexible, while a failed motor's bounds are set to **zero**, disabling it. This allows the controller to **adapt in real-time** and **optimize thrust** for the remaining motors.

4.2.3 Thrust Mapping

Another challenge we encountered was **mapping the thrust value** generated by our controller to the appropriate value that should be published to the ROS topic to achieve the desired thrust. The issue stems from the operational constraints of the drone's propulsion system: when the drone is armed, the minimum **rotor speed** is **150 rad/s**, corresponding to **zero thrust**, and any rotor speed below **150 rad/s** also produces no thrust. Furthermore, the system has an upper bound on rotor speed at **1000 rad/s**. To address this, we referred to the function used in the PX4 source code that maps the thrust to a normalized range of [0,1]. We then post-processed the controller output accordingly and published the resultant value to the appropriate ROS topic. The mapping function was found out to be

$$\boxed{\text{Normalized Value} = \frac{T/k + 150}{1000}}$$

where **T** is thrust and **k** is scaling factor found from model sdf files

4.2.4 Weights Fine-tuning

A significant challenge encountered during the implementation was the determination of the **cost matrix** utilized in the calculations. Weight parameters that had produced acceptable performance in Python simulations were found to be inadequate during testing within the PX4 and Gazebo environments. To optimize the matrix, systematic experimental analyses were conducted through iterative test runs in Gazebo. The primary issue observed was an excessive escalation in angular velocities, particularly along the x and y axes, resulting in divergence during the simulations. To address this, the cost matrix weights corresponding to angular velocities in the x and y directions were increased. Conversely, the cost associated

with angular velocity along the **z-axis** was significantly reduced due to the impracticality of controlling it.

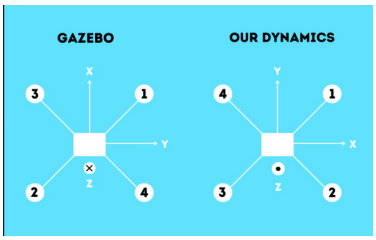


Figure 13: Comparative analysis of dynamics in Gazebo and the dynamics modeled in our study

5 Future Scope and Recommendations

5.1 Improving Observer Based detection

- **Observer Integration**: The observer can be pipelined with a model estimation application that outputs the moment of inertia matrix, thereby automating the manual calculation of MOI for drones carrying variable payloads.
- **Actuator Fault Estimation**: Complete actuator failures can be easily estimated. Partial actuator failures present greater challenges; hence, the observer can be modified to output actuator fault offset estimates for the detection and compensation of partial actuator faults.
- **Multi-Actuator Fault Handling**: Multi-actuator faults are yet to be accounted for, and the observer will fail in these scenarios. Traditional techniques will also fail as the system state depends on the order of actuator failures.
- **Wind-Induced Faults**: Failures caused by high-speed wind will also be detected by the observer as actuator faults. The observer can be modified to address wind and actuator faults separately.
- **Side Channel Detection**: Side channel or passive detection is considered cutting-edge. The measurement of emf responses from high-frequency PWM signals due to EMI over ESC supply lines can provide insights into whether a motor has failed.

5.2 Improving Controller System

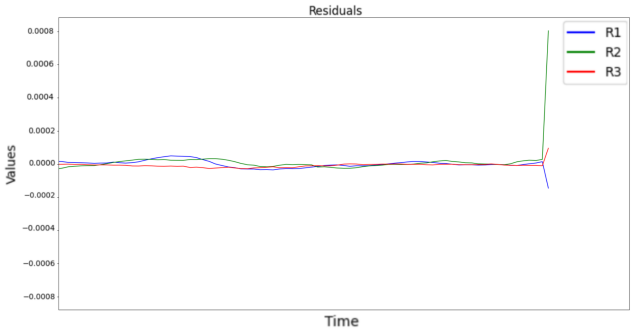
- **Current Weight Tuning for Agile Trajectories:**
  - The tuning of weights for the three-motor configuration requires further optimization.
  - Consistency in following agile trajectories is lacking, particularly for reliable Return-to-Home (RTH) pathways.
  - The system has achieved trajectory-following in isolated cases but requires repeatability and stability.
- **Key Integration Areas for the PS System:**
  - Enhancing Failure Detection Mechanisms.
  - Implementing Seamless Controller Switching.
  - Achieving Stable Hover Control.
  - Executing a Reliable Return-to-Home (RTH) Maneuver.
- **Latency in Failure Detection:**
  - Latency in failure detection results in non-convergent solutions in the Nonlinear Model Predictive Controller (NMPC) for the three-motor configuration.
  - Addressing this issue is critical for improving system stability and performance.

5.3 Hardware Implementation

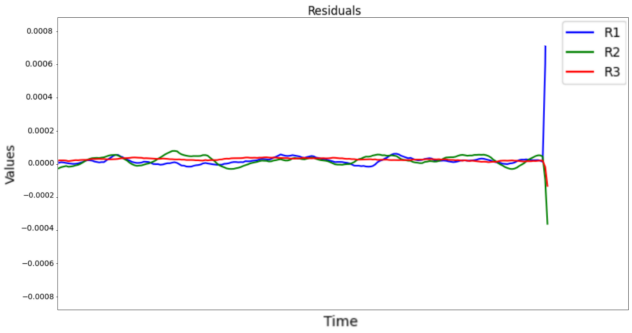
- **ESCs with Feedback for Motor Failure Detection:** The implementation of ESCs with feedback has shown significant potential in enhancing motor failure detection, as it enables quicker identification of failure and allows the quadrotor to remain more stable. While this approach demonstrated promising results during testing, it was not implemented due to regulatory restrictions.
- **Optical Flow Sensor for Stable Landing:** An optical flow sensor can be implemented such that the quadrotor can be more stable during landing.
- **Vision-Based Failsafe Algorithm Using Event Camera:** To enhance the positioning of a quadrotor in GPS-denied environments, the use of an Event Camera is being considered for a Vision-Based Failsafe Algorithm. The Event Camera offers significant advantages over conventional cameras, including the absence of motion blur and a higher dynamic range, which facilitates more precise tracking of the drone's position by monitoring surrounding features.

6 Appendix

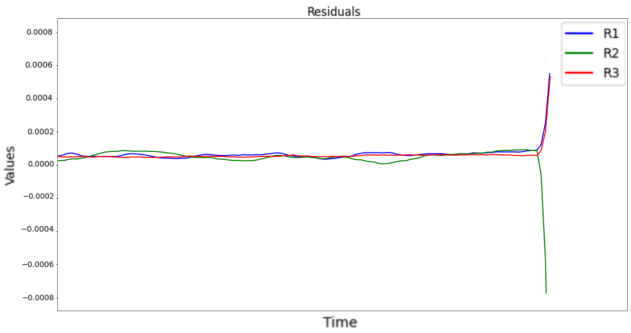
6.1 Residuals with Classical Observer based detection



(a) Residuals when Motor 2 is failed



(b) Residuals when Motor 3 is failed



(c) Residuals when Motor 4 is failed

6.2 Latency and Training Loss in Machine Learning based detection

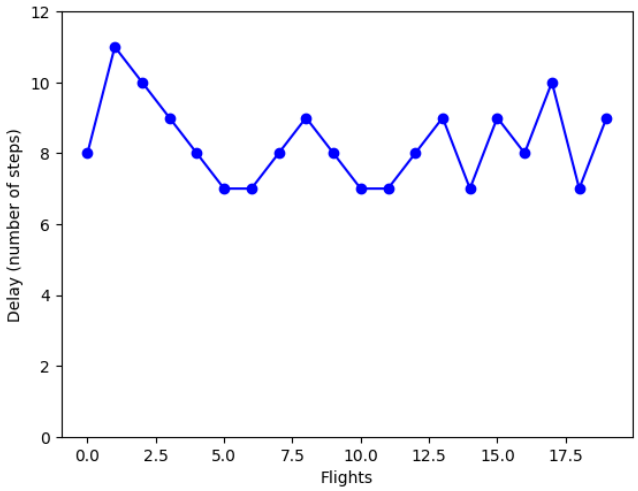


Figure 15: Latency Across Test Flights

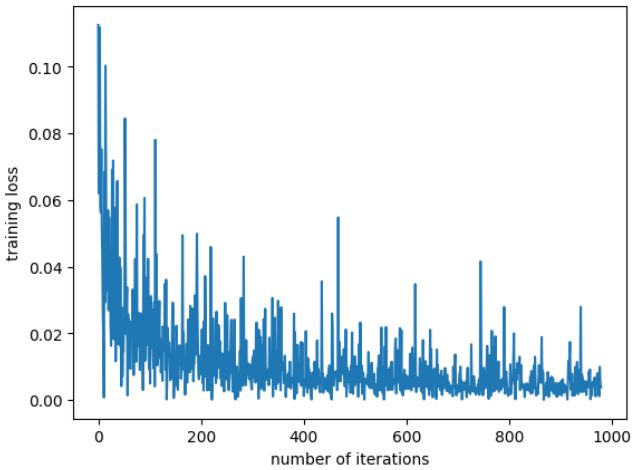


Figure 16: Training Loss vs Iterations

6.3 Controller Performance Metrics in Gazebo Simulation under various scenarios

6.3.1 Stable Hover

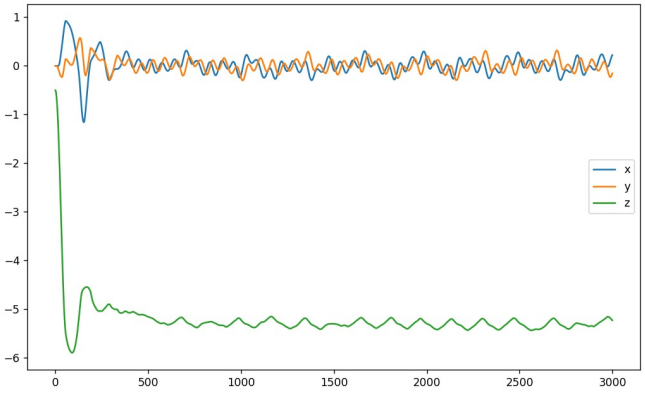


Figure 17: Position vs Time in steps of 10ms

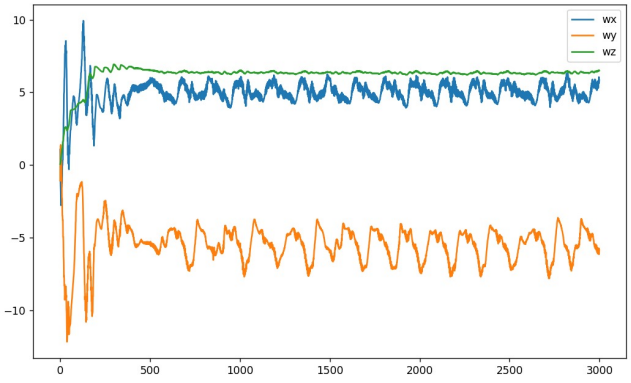


Figure 18: Angular Velocities vs Time in steps of 10ms

6.3.2 Controlled Landing

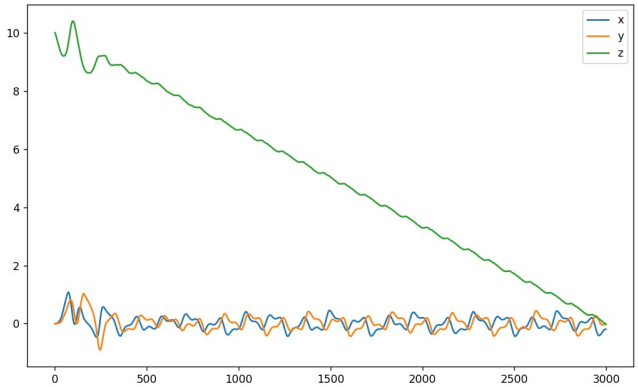


Figure 19: Position vs Time in steps of 10ms

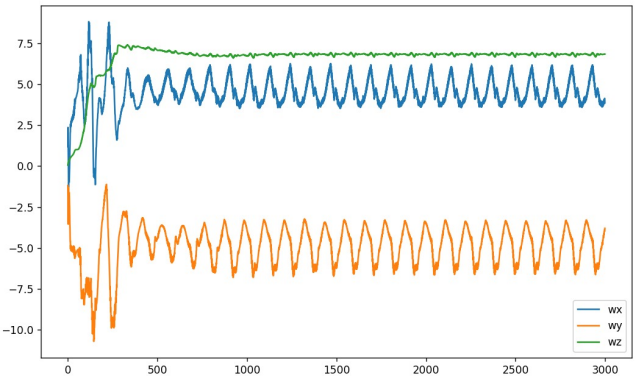


Figure 20: Angular Velocities vs Time in steps of 10ms

6.3.3 Return to Home

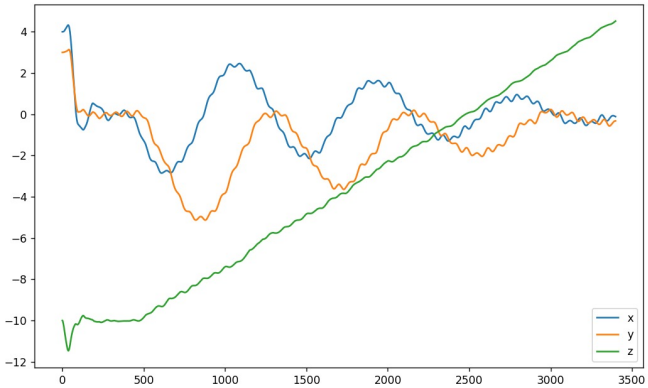


Figure 21: Position vs Time in steps of 10ms

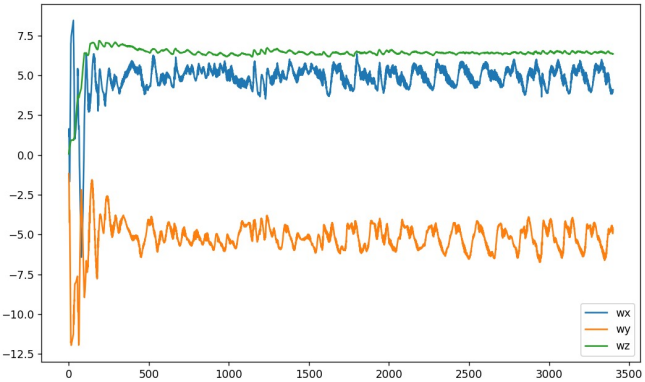


Figure 22: Angular Velocities vs Time in steps of 10ms

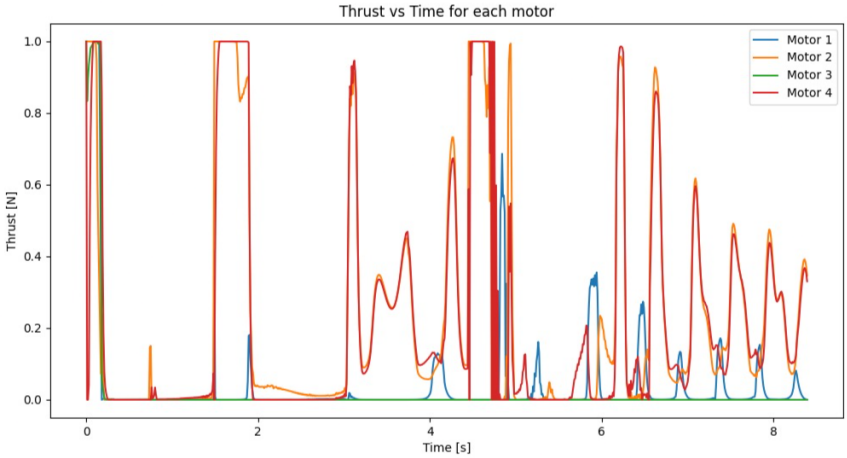


Figure 23: Thrust V/s Time for Each Motor before and after Motor Failure

References

[1] A. Freddi, S. Longhi, and A. Monteriù. "Actuator fault detection system for a mini-quadrotor". In: *2010 IEEE International Symposium on Industrial Electronics*. 2010, pp. 2055–2060. DOI: 10.1109/ISIE.2010.5637750.

[2] Ngoc Phi Nguyen and Sung Kyung Hong. "Sliding Mode Thau Observer for Actuator Fault Diagnosis of Quadcopter UAVs". In: *Applied Sciences* 8.10 (2018). ISSN: 2076-3417. DOI: 10.3390/app8101893.

[3] Zhaohui Cen, Hassan Noura, and Younes Al Younes. "Robust Fault Estimation on a real quadrotor UAV using optimized Adaptive Thau Observer". In: *2013 International Conference on Unmanned Aircraft Systems (ICUAS)*. 2013, pp. 550–556. DOI: 10.1109/ICUAS.2013.6564732.

[4] PX4 Development Team. *ROS Integration with PX4*. Available: https://docs.px4.io/main/en/ros2/user_guide.html. 2024.

[5] Gazebo Simulator. *Gazebo Installation*. 2024. URL: https://gazebosim.org/home.

[6] Open Robotics. *ROS 2 Documentation*. 2024. URL: https://docs.ros.org/en/humble/.

[7] PX4 Development Team. *Hello Sky Module*. Accessed: 2024-12-04. 2023. URL: https://docs.px4.io/main/en/modules/hello_sky.html.

[8] Venkata Sadhu, Saman Zonouz, and Dario Pompili. "On-board Deep-learning-based Unmanned Aerial Vehicle Fault Cause Detection and Identification". In: (2020). Available: https://arxiv.org/abs/2005.00336.

[9] Xiang Zhang et al. "Fault Detection and Identification Method for Quadcopter Based on Airframe Vibration Signals". In: *Sensors (Basel)* 21.2 (Jan. 2021), p. 581. DOI: 10.3390/s21020581.

[10] Sihao Sun et al. "Upset Recovery Control for Quadrotors Subjected to a Complete Rotor Failure from Large Initial Disturbances". In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020, pp. 4273–4279. DOI: 10.1109/ICRA40945.2020.9197239.

[11] Vincenzo Lippiello, Fabio Ruggiero, and Diana Serra. "Emergency Landing for a Quadrotor in Case of a Propeller Failure: A PID Based Approach". In: Oct. 2014. DOI: 10.1109/SSRR.2014.7017647.

[12] Fang Nan et al. "Nonlinear MPC for Quadrotor Fault-Tolerant Control". In: *IEEE Robotics and Automation Letters*. 7.2 (2022), pp. 5047–5054. DOI: 10.1109/LRA.2022.3154033.