Inter IIT Tech Meet 13.0

Position and Altitude Control of Quadrotor with Single Motor Failure

Midterm Report

Submitted by :
Team 30

Abstract

Presents detailed literature review,

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. The primary objective is to accurately detect the faulty motor, reducing both false negatives and false positives, ensuring timely detection. Following this we aim to stabilize the quadrotor and adjust control input to the remaining three motors to facilitate movement to enable safe landing or custom trajectory following.

2 Motor Failure Detection

We proceeded with two approaches for isolating the motor failure. The results of both approaches are provided below.

2.1 Classical Dynamics Approach

An observer system was formulated and implemented to estimate state residuals. These state residuals map in different combinations to sets of actuator failures.

2.1.1 Literature Review

In our research, we reviewed several studies that address single motor failure detection in Unmanned Aerial Vehicles (UAVs). Notably, the work by A. Freddi et al. in this system has been discussed in [1]. This paper introduces an actuator fault detection system that employs a **Thau observer** to detect faults by generating **Residuals**, which help identify if and when a fault occurs.

Residuals are algebric differences of **estimated** state variables and **measured** state variables. In normal operation, these residuals remain near zero. When a fault occurs, these residuals quickly blow up and help isolate actuator fault diagonal.

A significant study in fault detection and identification (FDI) for quadcopters is presented by Ngoc Phi Nguyen et al. in [2]. This paper provides a robust method specifically for detecting motor failures in quadcopter UAVs. It introduces the **Sliding Mode Thau Observer (ASMTO)**, which allows for real-time detection and quantification of motor faults by observing deviations in the system's expected behavior. The ASMTO can identify not only the presence of a motor fault but also estimate its magnitude and track its progression over time, while rejecting external distubances.

Another cutting edge paper is presented by Zhaohui Cen et al. in [3]. This paper provides a comprehensive approach to rotor failure detection for quadrotor UAVs by using an **Adaptive Thau Observer (ATO)**. Similar to **ASMTO**, this also identifies and isolates motor faults while advancing it with estimating fault severities. This is a more robust technique as compared to ASMTO as it takes care of unmodelled dynamics by cascading more optimised observers with the ATO.

One more relevant paper we found was by Khalid Mohsin Ali et al. in [4]. There was a lot of non-uniformity in literature on usage of **CROSS** or **PLUS** configuration of quadrotors. We used this paper to stick to **CROSS** configuration as it matched with **PX4** conventions. This necessitated a conversion of residuals back to **PLUS** frame, as only then they are useful in isolating diagonal faults.

2.1.2 System Description

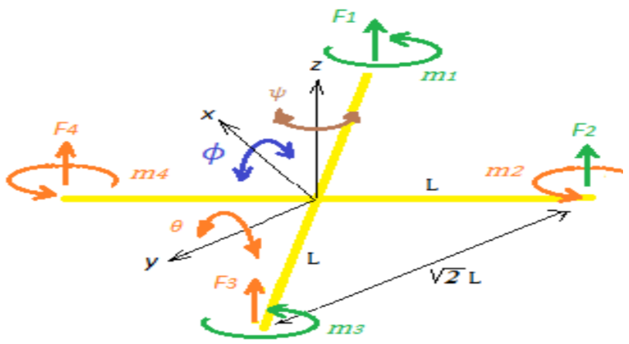


Figure 1: Schematic of the geometric configuration of the quadcopter

The control variables can be described as

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

$\Omega_1, \Omega_2, \Omega_3, \Omega_4$ are the speeds of the four motors. And Ω_r is the residual rotational velocity:$\Omega_r = -\Omega_1 + \Omega_2 - \Omega_3 + \Omega_4$
(2)

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

$$\ddot{\phi} = \frac{\dot{\theta}\dot{\psi}(I_y - I_z) - J_r\dot{\theta}\Omega_r + u_2}{I_x} \tag{3}$$

$$\ddot{\theta} = \frac{\dot{\phi}\dot{\psi}(I_z - I_x) + J_r\dot{\phi}\Omega_r + u_3}{I_y} \tag{4}$$

$$\ddot{\psi} = \frac{\dot{\phi}\dot{\theta}(I_x - I_y) + u_4}{I_z} \tag{5}$$

(6)

where $I_x, I_y,$ and I_z represent the moments of inertia along the $x, y,$ and z directions, respectively. (7)

By Defining the state vector

$$x^T = \begin{bmatrix}\varphi & \theta & \psi & \dot{\varphi} & \dot{\theta} & \dot{\psi}\end{bmatrix},$$

control input vector

$$u^T = \begin{bmatrix}u_2 & u_3 & u_4\end{bmatrix},$$

and output vector

$$y^T = \begin{bmatrix}\varphi & \theta & \psi & \dot{\varphi} & \dot{\theta} & \dot{\psi}\end{bmatrix},$$

The above equations can be described in the state equation as

$$\begin{cases} \dot{x}(t) = Ax(t) + Bu(t) + h(x,u) \\ y = Cx(t) \end{cases} \tag{6}$$

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

$$B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1/I_x & 0 & 0 \\ 0 & 1/I_y & 0 \\ 0 & 0 & 1/I_z \end{bmatrix},$$

$$C = I_{6\times 6} \quad and$$

$$h(x,u) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ (\dot{\theta}\dot{\psi}(I_y - I_z) - J_r\dot{\theta}\Omega_r)/I_x \\ (\dot{\varphi}\dot{\psi}(I_z - I_x) + J_r\dot{\varphi}\Omega)/I_y \\ \dot{\varphi}\dot{\theta}(I_x - I_y)/I_z \end{bmatrix}.$$

2.1.3 Setting up the Thau's Observer

A stable observer for the system has the form

$$\begin{aligned} \dot{\hat{x}}(t) &= A\hat{x}(t) + Bu(t) + h(\hat{x}(t), u(t)) + K(y(t) - \hat{y}(t)) \\ \hat{y}(t) &= C\hat{x}(t) \end{aligned}$$

where $\boldsymbol{K}$ is the observer gain matrix.
Then, we used Runge-Kutta 4th order (RK4) equations to get $\hat{x}(t)$ from $\dot{\hat{x}}(t)$:

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

2.1.4 Calculating the residuals

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

2.2 Machine Learning Approach

2.2.1 Literature Review

In our research, we reviewed several studies that address single motor failure detection in Unmanned Aerial Vehicles (UAVs). Notably, the work by Sadhu et. al. in [5] presents a deep learning-based approach. They employed a two-step methodology in which the identification/classification phase occurs only after detecting anomalous behavior in sensor data.

Their proposed method includes a Convolutional Neural Network (CNN) and Bi-directional Long Short Term Memory (BiLSTM) deep neural network-based autoencoder to detect faults or anomalous patterns, followed by a CNN-LSTM deep network for classification and identification.

In our implementation of this method, we created a sliding window of 20 timestamps with a stride of 1 to pass to the network. This setup provided an inference time of 20 ms per window. However, it initially achieved only 50% accuracy in the first window, which improved to 95% after five windows. This led to latency in our detection approach, making it unsuitable for our requirements, and ultimately, we decided not to adopt this method.

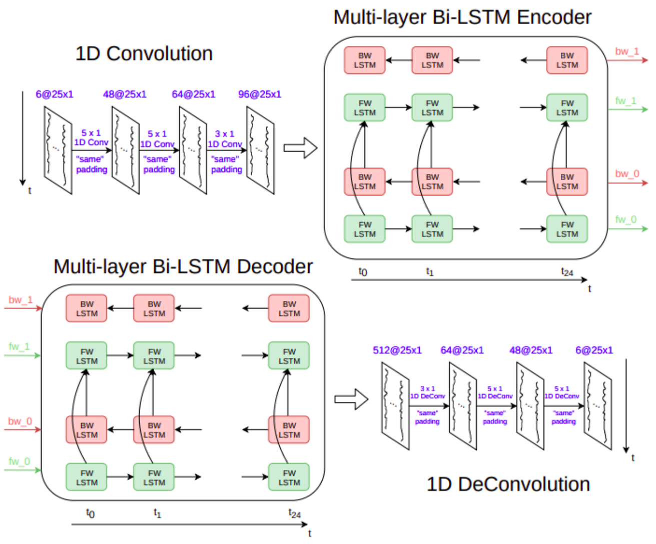

1D Convolution
6@25x1
48@25x1
64@25x1
96@25x1
5 x 1 1D Conv "same" padding
5 x 1 1D Conv "same" padding
3 x 1 1D Conv "same" padding
Multi-layer Bi-LSTM Encoder
BW LSTM
FW LSTM
bw_1
fw_1
bw_0
fw_0
t0
t1
t24
Multi-layer Bi-LSTM Decoder
512@25x1
64@25x1
48@25x1
6@25x1
3 x 1 1D DeConv "same" padding
5 x 1 1D DeConv "same" padding
5 x 1 1D DeConv "same" padding
1D DeConvolution
t

Figure 2: Architecture of CNN Bi-LSTM

Zhang et al. present a notable study on fault detection and identification (FDI) for quadcopters in Fault Detection and Identification Method for Quadcopter Based on Airframe Vibration Signals [6].Their FDI method relies on analyzing the vibration signals from the quadcopter airframe. This approach comprises vibration data acquisition, data preprocessing, feature extraction, and LSTM-based FDI model training. Initially, accelerometer sensors capture the airframe vibration data during quadcopter flight. The collected data sets are divided into K subsets corresponding to K health states, yielding data sets D_i $(i = 1, 2, \dots, K)$. Each data set is further divided into subsets based on 1-second intervals, producing processed data sets d_i $(i = 1, 2, \dots, K)$.

Wavelet packet decomposition is then applied for feature extraction, resulting in feature vectors for each data set, represented as θ_i $(i = 1, 2, \dots, K)$. These feature vectors are then used to train the LSTM model, enabling quadcopter FDI based on airframe vibration signals.

The N-axis airframe vibration signal of the quadcopter in data set d_i is represented as:

$$\delta^i = [\delta_1^i, \delta_2^i, \dots, \delta_q^i] \in \mathbb{R}^{N \times \lambda \times q}. \tag{8}$$

This approach achieved an accuracy of 96% in detecting motor failures. However, due to the computationally intensive wavelet transformation, the model introduced significant latency, which could affect its performance in real-time applications.

2.2.2 Our Approach

After analyzing various approaches in the literature, we implemented a lightweight Recurrent Neural Network (RNN) model for motor failure detection. This model consists of one input layer, two hidden layers, and an output layer, as shown in Figure 3. The model is designed to be highly efficient, with only 31,000 trainable parameters, resulting in a compact size of 0.13 MB.

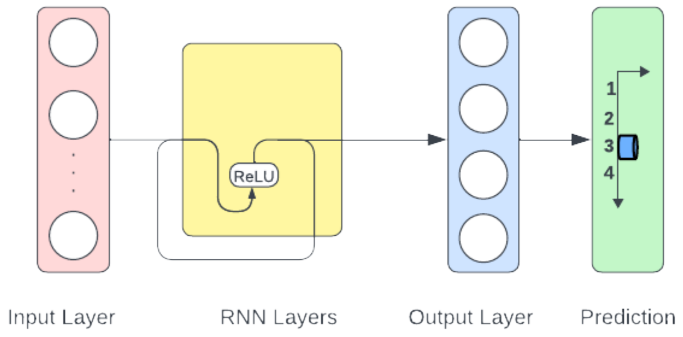


Figure 3: Model Layers

The use of an RNN is advantageous for this application because RNNs are well-suited for sequential data processing. In motor failure detection, sensor data arrives as a continuous stream of readings over time, creating a sequential dataset. Unlike traditional neural networks, RNNs are designed to retain information from previous time steps, making them ideal for modeling temporal dependencies within the data. The major issue of vanishing gradients in Recurrent Neural Networks (RNNs) is actually advantageous for our model. In this context, we do not want older information about the drone to stay in the network. The model inherently clears less recent data from the drone by vanishing it in the forward pass.

Our model is trained on 10 features derived from IMU sensor readings: Angular Acceleration X, Angular Acceleration Y, Angular Acceleration Z, Linear Acceleration X, Linear Acceleration Y, Linear Acceleration Z, Orientation X, Orientation Y, Orientation Z, and Orientation W. These features provide a comprehensive representation of the quadcopter's motion and orientation, allowing the model to detect changes that may signify motor failure. Each input to the model has a shape of $(1, 10)$, corresponding to these 10 parameters, and the model predicts which motor has failed.

By capturing the sequential patterns in these IMU readings—such as accelerations and orientations over time—the RNN can recognize subtle temporal changes that may indicate motor anomalies. This capability to model time-dependent variations in the data enables the RNN to effectively identify motor failures based on the sensor data stream.

2.2.3 Training

To train the model, we first created a comprehensive dataset that includes all 10 features mentioned above in sequential order, along with an additional column indicating the actual failed motor for each timestamp. This column serves as the target label for training. However, directly predicting the index of the failed motor could lead the network to establish a topology among the motor indices. Therefore, the model was trained to predict a one-hot vector, where the position of 1 signifies the index of the failed motor.

For our dataset collection, we used Software-In-The-Loop (SITL) simulation, which let PX4 run as if it were controlling a real drone, while interacting within Gazebo Classic's virtual world. This setup allowed PX4 to mimic real-world sensor readings and responses to control inputs. With MAVROS serving as the bridge between ROS and PX4, we could seamlessly gather live data from topics like `/mavros/imu/data`, capturing details such as the drone's acceleration and orientation. This continuous flow of IMU data was essential to analyze how the drone responds to various flight conditions.

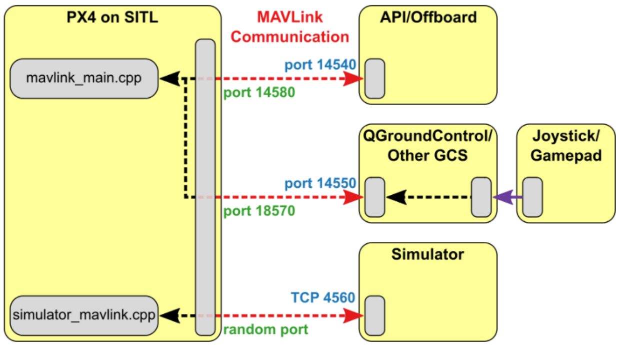


Figure 4: Mavlink Connection

To track motor failures, we connected to the `/motor_failure/motor_number/` topic, which indicated any motor that stopped working during the simulation. This was powered by a custom motor failure plugin in Gazebo that simulated real-time motor stoppages. With MAVROS enabling communication across ROS, PX4, and Gazebo, we built a robust setup to study the drone's behavior under failure scenarios. Further details on the motor failure plugin implementation are shared in the following section.

We also subscribed to the `/mavros/local_position/pose` topic to monitor the quadcopter's Z-axis position. Tracking the Z-position is essential because we stop data collection if the drone's altitude falls below 0.5 meters. This threshold helps prevent erroneous data from being recorded if the drone crashes, which could mislead the model during training by causing it to associate crashes with motor failure.

Overall, this dataset accurately captures the IMU readings and motor failure labels needed to train the model, allowing it to recognize and predict motor failures based on real-time sensor inputs. To train the model, we recorded approximately 40 flights for the training dataset and 20 flights for testing purposes. The collected data was converted into a CSV file for structured input.

For model training, we feed each row of the CSV file as an input vector of shape $(1, 10)$, corresponding to the ten features captured from the IMU sensors. Along with this input vector, we provide a label indicating the motor that has failed for that specific timestamp. As the model processes each sequential input, it updates its memory, allowing it to learn temporal patterns and correlations that may indicate motor failure.

This approach enables the model to effectively recognize and predict motor failures based on the sequential data obtained from the IMU sensors, leveraging the recorded flight data for enhanced accuracy.

We calculated the model's accuracy at the point of actual motor failure. For each of the 20 test files, we evaluated whether the model correctly identified the motor failure at the precise timestamp it occurred. Our results demonstrated that the model achieved an accuracy of over 99%, indicating that it correctly detected the failed motor within a tolerable delay in 99% of the times. Figure 5 shows the number of steps taken by the model to predict the failure across the test flights. On average, 9 steps are taken by the model to detect the failure where each step at a refresh rate of 100 Hz gives an overall inference time of around 0.09 s. This high accuracy underscores the model's reliability in real-time motor failure detection.

Figure 6 demonstrates the reduction of the mean squared loss between the target and the predicted one-hot vector. The spikes in the loss represents the beginning of a new test file.

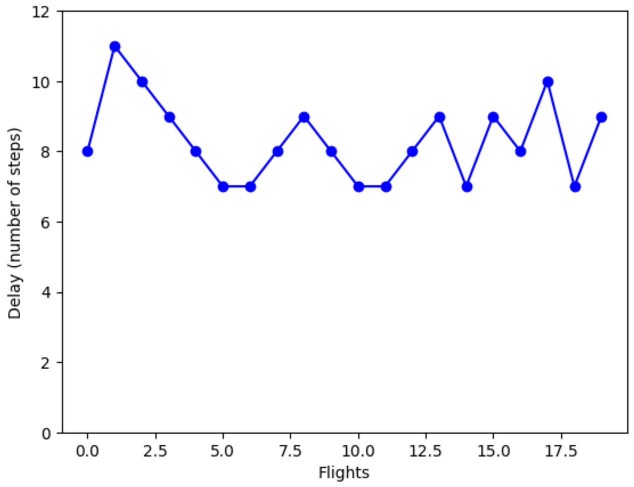


Figure 5: Latency across test flights

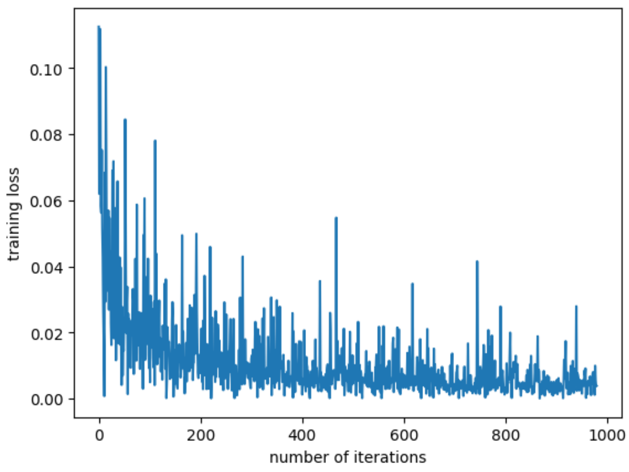
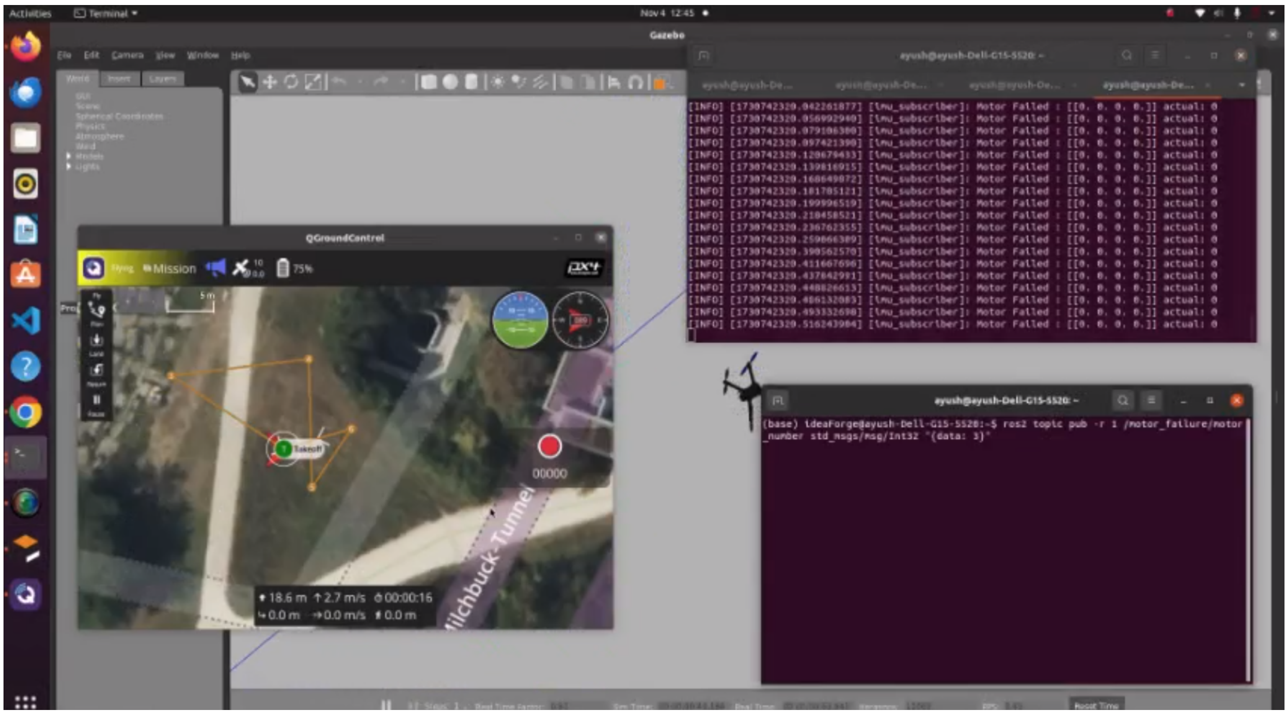


Figure 6: Training Loss vs Iterations



```
[INFO] [1730742320.042261877] [imu_subscriber]: Motor Failed : [[0. 0. 0. 0.]] actual: 0
[INFO] [1730742320.056992940] [imu_subscriber]: Motor Failed : [[0. 0. 0. 0.]] actual: 0
[INFO] [1730742320.079106300] [imu_subscriber]: Motor Failed : [[0. 0. 0. 0.]] actual: 0
[INFO] [1730742320.097421390] [imu_subscriber]: Motor Failed : [[0. 0. 0. 0.]] actual: 0
[INFO] [1730742320.120679433] [imu_subscriber]: Motor Failed : [[0. 0. 0. 0.]] actual: 0
[INFO] [1730742320.139816915] [imu_subscriber]: Motor Failed : [[0. 0. 0. 0.]] actual: 0
[INFO] [1730742320.160649872] [imu_subscriber]: Motor Failed : [[0. 0. 0. 0.]] actual: 0
[INFO] [1730742320.181785121] [imu_subscriber]: Motor Failed : [[0. 0. 0. 0.]] actual: 0
[INFO] [1730742320.199996519] [imu_subscriber]: Motor Failed : [[0. 0. 0. 0.]] actual: 0
[INFO] [1730742320.210458521] [imu_subscriber]: Motor Failed : [[0. 0. 0. 0.]] actual: 0
[INFO] [1730742320.236762355] [imu_subscriber]: Motor Failed : [[0. 0. 0. 0.]] actual: 0
[INFO] [1730742320.259866309] [imu_subscriber]: Motor Failed : [[0. 0. 0. 0.]] actual: 0
[INFO] [1730742320.390562570] [imu_subscriber]: Motor Failed : [[0. 0. 0. 0.]] actual: 0
[INFO] [1730742320.411667696] [imu_subscriber]: Motor Failed : [[0. 0. 0. 0.]] actual: 0
[INFO] [1730742320.437842991] [imu_subscriber]: Motor Failed : [[0. 0. 0. 0.]] actual: 0
[INFO] [1730742320.448826613] [imu_subscriber]: Motor Failed : [[0. 0. 0. 0.]] actual: 0
[INFO] [1730742320.486132083] [imu_subscriber]: Motor Failed : [[0. 0. 0. 0.]] actual: 0
[INFO] [1730742320.493332698] [imu_subscriber]: Motor Failed : [[0. 0. 0. 0.]] actual: 0
[INFO] [1730742320.516243904] [imu_subscriber]: Motor Failed : [[0. 0. 0. 0.]] actual: 0
```

```
(base) ideaForge@ayush-Dell-G15-5520:~$ ros2 topic pub -r 1 /motor_failure/motor
_number std_msgs/msg/Int32 "{data: 3}"
```

Figure 7: Testing on Live Data

3 Recovery and Post-Failure Control

After completing the motor failure detection process, we can proceed to develop our post-detection stabilization algorithm. Upon detecting a motor failure, the algorithm's priority will be to capture and store the quadrotor's reference state immediately before the failure. This reference state acts as a target for recovery, enabling the quadrotor to return to its prior position. The algorithm will then work to stabilize the quadrotor, achieving a controlled hovering state at this captured position. By prioritizing a return to the pre-failure reference point, we aim to minimize deviation and ensure safety in response to unexpected motor failure.

3.1 Literature Review

3.1.1 Upset Recovery Control for Quadrotor

For our research, we consulted the study *Upset Recovery Control for Quadrotors Subjected to a Complete Rotor Failure from Large Initial Disturbances* by Sun et al. [7], which addresses quadrotor fault recovery under severe disturbances.

The combination of Incremental Nonlinear Dynamic Inversion (INDI) and Non-Singular Terminal Sliding Mode Control (NTSMC) offers a robust solution for fault-tolerant control, especially suited for systems exposed to large and unexpected disturbances. INDI dynamically adjusts control inputs in response to detected faults, incrementally compensating for system deviations in real time. This allows for gradual corrective action that smoothly addresses issues without necessitating sudden changes, making INDI especially useful in situations where stability is crucial despite fluctuations. NTSMC, on the other hand, is designed to be resilient against significant disturbances, using terminal sliding surfaces to guide the system state toward a desired target within a finite timeframe. This finite-time convergence ensures that even under considerable disruptions, the system remains stable and can recover quickly, effectively rejecting faults and uncertainties to achieve reliable performance.

One of the primary strengths of the INDI + NTSMC approach is its effectiveness in handling substantial disturbances. NTSMC's inherent robustness allows it to drive the system to a terminal sliding surface that maintains stability, even when exposed to large, sudden faults. This is particularly beneficial in critical applications where system reliability must be preserved despite unpredictable conditions. Meanwhile, INDI provides incremental and adaptive compensation, allowing the system to adjust to faults gradually. This measured response prevents abrupt control actions, which could destabilize the system under high-stakes conditions, and instead provides a steady correction that sustains performance over time.

However, the approach also has certain limitations. While INDI excels in counteracting larger faults, its capacity to manage smaller disturbances or nuanced variations in system dynamics is limited. This reduced precision can affect fault compensation accuracy, particularly in scenarios requiring fine-grained control where even minor deviations matter. Additionally, NTSMC's aggressive nature, optimized for large disturbance rejection, may result in overcompensation when dealing with minor faults. This overreaction can cause unnecessary oscillations or degrade performance in instances where a more subtle control action would suffice, making it less effective in cases where precision over brute-force correction is needed. Finally, the INDI + NTSMC framework is inherently reactive: it responds to faults only after they occur, lacking predictive capabilities that could optimize control actions based on anticipated disturbances. This reactive approach limits adaptability in highly dynamic environments, where a forward-looking, anticipatory control strategy could improve performance by proactively addressing potential issues before they impact the system.

3.1.2 Emergency Landing for a Quadrotor using PID based approach

We referred the research paper *Emergency Landing for a Quadrotor in Case of a Propeller Failure: A PID-Based Approach* by Lippiello et al. [8], which was based on a PID-based approach to find solution for motor failure in a quadrotor during flight. When a motor failure occurs, maintaining the stability and control of the drone becomes very difficult . This study describes a PID controller approach of adjusting the control inputs of other three rotors to attain stable hover and emergency landing.

The research shows the effectiveness of PID control in stabilizing the quadrotor when one motor failed. The PID controller continuously adjusts the thrust and speed of the remaining motors to maintain the imbalance caused by the motor failure. It is shown in paper that it is theoretically possible for a birotor to follow a given path to a certain degree of accuracy, allowing for safe landing, even with a failure along diagonal of motor. This approach is extremely useful in situations where there is no immediate risk of completely loosing control

However this approach has a problem. The drone's overall ability to lift is greatly reduced since the work done by four motors is now carried out by only two motors. The remaining motors must compensate for the loss of thrust, which increases their workload and reduces the drone's ability to maintain altitude. This means the drone might not be able to clear obstacles or reach the intended emergency landing zone, limiting its functionality.

3.1.3 Non Linear MPC for Quadrotor Failure

To achieve robust stabilization, recovery, and controlled landing for the quadrotor, we integrated a Nonlinear Model Predictive Controller (NMPC) that utilizes a custom-defined cost function, developed by Sihao Sun in [9]This advanced approach allows us to address the complexities of post-failure flight by fully incorporating the nonlinear dynamics of a damaged quadrotor system.

Unlike conventional methods that often depend on linear approximations or omit motor input constraints within the outer control loop, NMPC maintains a comprehensive perspective on system behavior. By accounting for both the nonlinear dynamics and the specific thrust limitations of each motor, NMPC provides a more precise and reliable response in failure scenarios. This holistic design allows the controller to adapt seamlessly to real-time system changes and constraints.

The implementation of NMPC demonstrated promising results, even under extreme conditions where traditional controllers tend to struggle with stability. With this approach, we prioritize return to the pre-failure reference position first, followed by a achieving a stable hover at the same position, and ultimately, a controlled descent and safe landing. The nuanced dynamics captured within this NMPC framework enable the quadrotor to respond flexibly and securely to sudden motor failures, offering an effective path for managing failure recovery while enhancing safety and control during challenging flight conditions.

3.2 Our Approach

3.2.1 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, we can predict its motion. Our model considers the following state variables :

3.2.2 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

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 1: Drone State and Control Variables

$\boldsymbol{\omega} = [\omega_x, \omega_y, \omega_z]^T$ represent the angular velocity in the body frame. The dynamic model is given by:

$$\dot{\mathbf{p}} = \mathbf{v}, \tag{9}$$

$$\dot{\mathbf{q}} = \frac{1}{2}\mathbf{q} \circ \begin{bmatrix} 0 \\ \boldsymbol{\omega} \end{bmatrix}, \tag{10}$$

$$\dot{\mathbf{v}} = \mathbf{q} \otimes \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix} - \mathbf{g}, \tag{11}$$

$$\dot{\boldsymbol{\omega}} = \mathbf{I}^{-1} \left(\boldsymbol{\tau} - \boldsymbol{\omega} \times (\mathbf{I}\boldsymbol{\omega}) + \boldsymbol{\tau}_{\text{ext}}\right), \tag{12}$$

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.

~~Additionally, motor dynamics are modeled as a first-order system: $\dot{T}_i = \frac{1}{\sigma}(u_i - T_i), \quad i = 1, 2, 3, 4,$ (13) where σ is the motor time constant, and u_i is the thrust command for rotor i.~~

3.2.3 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:

$$\min_{\mathbf{u}_{k:k+N-1}} \mathbf{y}_N^T \mathbf{Q}_N \mathbf{y}_N + \sum_{i=k}^{k+N-1} \left(\mathbf{y}_i^T \mathbf{Q} \mathbf{y}_i + \mathbf{u}_i^T \mathbf{R} \mathbf{u}_i\right), \tag{14}$$

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 predictive process. By enforcing these penalties at each stage—not only at the endpoint—the controller is encouraged to take corrective actions that maintain alignment with desired values over the entire prediction horizon. The stage cost typically assigns a penalty for discrepancies between the actual and target values of both the system's state and control inputs at intermediate steps. This way, the controller actively works to keep the system on track at every point in the horizon, promoting a smooth progression toward the desired outcome. By prioritizing steady correction rather than only focusing on the final state, the stage cost fosters improved stability and reliability, especially in dynamic or disrupted conditions.

In our NMPC framework, 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. By applying this cost only at the conclusion, it acts as an endpoint anchor, guiding the system toward stability and optimality as it approaches the final predicted state. The terminal cost, in essence, reinforces the goal of reaching a specific target state by the end of the prediction, complementing the stage costs that shape the system's trajectory along the way. Together, these costs work in a balance: the stage cost ensures steady adherence to the desired trajectory at each incremental step, while the terminal cost reinforces a successful convergence to the target state at the horizon's end. This synergy between the stage and terminal costs is essential, fostering both the stability and precision needed to achieve optimal control over the quadrotor's recovery and stabilization after motor failure.

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

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

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 and nuances of the drone's behavior. This approach allows us to incorporate the inherent nonlinearities of quadrotor flight, capturing the intricate interactions between variables that are often simplified or overlooked in linear models. As a result, our model is more accurate and responsive to real-world conditions, enabling a more effective control strategy that can handle the unique challenges of flight dynamics without sacrificing precision or robustness.

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

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. This prevents the controller from generating unattainable thrust values, which would otherwise risk system instability or failure. Consequently, these constraints contribute to a reliable and practical model that aligns closely with the quadrotor's operational limits, enhancing safety and performance during real-world implementation.

The $\mathbf{Q}$, $\mathbf{Q}_N$, and $\mathbf{R}$ are positive-definite weight matrices, which were optimally obtain as follows : *insert gazebo weightage values*.

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. This approach allows our model to account for the motor's non-functioning state accurately within the control framework, ensuring that no thrust is allocated to the failed motor. This simple yet effective constraint adjustment allows us to simulate failure conditions and test the quadrotor's response under realistic, constrained scenarios.

3.2.4 Attitude Error and Cost Vector

We compute the error which and the cost vector $\mathbf{i}\mathbf{y}_{\mathrm{i}}$ in this subsection. 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 $\mathbf{q}_{\mathrm{ref}}$ 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 (). Once we have this we perform an split in the quaternion error. We divide it into the $\mathbf{q}_{\mathrm{xy}}$ and $\mathbf{q}_{\mathrm{z}}$. This is because $\mathbf{q}_{\mathrm{z}}$ can never be controlled properly with 3 motors so we must separate the controllable part of the quaternion. Here $\mathbf{q}_{\mathrm{z}}$ is obtained by isolating the scalar and the z component of the quaternion error we obtained in the earlier step. To compute $\mathbf{q}_{\mathrm{xy}}$ term, we perform quaternion multiplication of the q with $\mathbf{q}_{\mathrm{ref}}$ inverse.

Another challenge we faced was defining a realistic reference angular velocity, $\mathbf{w}_{\mathrm{ref}}$, for the system. Setting $\mathbf{w}_{\mathrm{ref}} = \mathbf{0}$ 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, $\mathbf{w}_{\mathrm{ref,z}}$ 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 $\mathbf{w}_{\mathrm{ref}}$ 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 the loss of one motor.

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 control actions. 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. By maintaining these zero references for non-position states, we create a stable environment where the drone focuses solely on achieving and maintaining the target position, ensuring minimal deviation and a smooth, controlled hover.

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.

3.2.5 Implementation of NMPC using CasADi

Initially, we attempted to implement the NMPC for quadrotor recovery entirely using CasADi, leveraging its symbolic computation capabilities for both model formulation and optimization. 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.

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. Each optimization iteration required extensive computation, particularly as we handled multiple constraints and adjusted the thrust outputs dynamically in response to motor failure.

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. It supports a variety of advanced numerical techniques which divides the prediction horizon into segments to speed up computation.

Upon switching to ACADOS, the 2-second latency vanished, and the NMPC was able to generate control inputs in real time, ensuring a prompt response to motor failure scenarios. This improvement allowed the quadrotor to maintain stability and trajectory tracking immediately after a motor failure was detected, with minimal deviation from the reference path.

The successful integration of CasADi for symbolic modeling and ACADOS for optimization proved to be a robust solution, combining 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.

3.2.6 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 operates by calculating the difference between the reference states and the current states of the system. Specifically, it takes the reference values, subtracts the actual state values, and squares the results. Squaring the differences serves two main purposes: it ensures that both positive and negative discrepancies are accounted for, and it amplifies larger deviations, making the function more sensitive to larger errors. 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. This approach ensures that the system behaves as close to the desired state as possible within the defined constraints.

The results obtained from this method demonstrated the system's ability to converge effectively toward the reference trajectory, optimizing performance by minimizing the cost function. This minimization helped in reducing deviations and maintaining stability in the control process, leading to an overall improved outcome in our optimization task.

~~Insert images from LINEAR_LS~~

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.

The EXTERNAL cost function offered increased flexibility and customization potential compared to `' '` `'LINEAR_LS'`, allowing us to incorporate more complex relationships and constraints directly into the cost structure. However, implementing it involved re-evaluating each parameter to ensure alignment with the new cost function. This process included recalibrating weights to accurately capture the importance of various state and control elements under the new configuration.

Below, we outline the key changes made to the solvers and weights, 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.

In our implementation of the NMPC controller for the quadrotor using `ACADOS`, we carefully configured several solver options to ensure high performance in real-time applications. The core of the NMPC optimization was handled by selecting an appropriate solver and integrator, which together allowed for efficient handling of the system's nonlinear dynamics and constraints.

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. The HPIPM solver is well-suited for problems with sparse structures and nonlinear constraints, like those encountered in NMPC. The solver efficiently computes the optimal control inputs while reducing computational complexity, which is critical for real-time applications.

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. It allowed us to predict the future states of the system over the prediction horizon, providing the necessary accuracy while maintaining low computational cost. This integrator was well-suited to handle the quadrotor's flight dynamics while ensuring fast computation of control inputs.

We configured the solver to use the `SQP_RTI` (Sequential Quadratic Programming with Real-Time Iteration) solver type. This method iterates on a sequence of quadratic approximations of the nonlinear optimization problem, progressively refining the solution at each timestep. The `Real-Time Iteration (RTI)` aspect ensures that updates to the solution are incremental, which allows for faster convergence and avoids the need for recomputing the entire optimization problem from scratch at every timestep. This iterative approach is key for achieving real-time performance in fast-moving systems such as a quadrotor.

Additionally, we selected the `Gauss-Newton` method for `Hessian` approximation. Calculating the exact Hessian matrix can be computationally expensive, so the `Gauss-Newton approximation` was used to reduce the complexity of the optimization problem. This method approximates the second-order derivatives, which significantly speeds up the solver without compromising on the quality of the solution.

These solver settings combined to form an efficient NMPC controller capable of real-time flight control, even under fault conditions such as motor failure. The integration of `ACADOS` with these solver configurations ensured that the optimization process was fast enough to allow for prompt control input generation, enabling the quadrotor to maintain stability and track its trajectory accurately.

3.2.7 Simulation and Testing

The CasADi-ACADOS based NMPC was tested in a simulated environment 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.

4 Challenges Ahead and Next Steps

A important step ahead is to test our simulation results on real hardware to see if our solution works as expected and is easy to use. This testing phase is non-trivial due to a non-ideal environment and risk to damaging our hardware during trials. However, real-world testing is essential to check if the system stays stable and can move as desired when a motor fails. Proving that our solution works well on hardware will show that the solution we have proposed is relevant to the industry.

5 Workspace Setup

We have utilized Docker to set up our entire workspace, enabling users across all versions of various operating systems to work in a consistent environment, eliminating code-based dependencies. All dependencies are documented in the code repository, along with the Docker setup described below:

```
# Use the base image for ROS Humble Desktop
FROM osrf/ros:humble-desktop

# Set build arguments
```

```
ARG USER=ideaForge
ARG DEBIAN_FRONTEND=noninteractive

RUN groupadd -r ${USER} && useradd -r -g ${USER} ${USER} && \
    mkdir -p /home/${USER} && \
    chown -R ${USER}:${USER} /home/${USER} && \
    echo ${USER}' ALL=(ALL) NOPASSWD: ALL' >> /etc/sudoers && \
    chmod 0440 /etc/sudoers

# Set the working directory
WORKDIR /home/${USER}

# Update package lists and install required ROS packages and tools
RUN apt-get update && \
    apt-get install -y \
    python3 \
    python3-pip \
    nano \
    git \
    wget \
    curl \
    ros-humble-gazebo-* \
    ros-humble-gazebo-ros-pkgs \
    vim && \
    rm -rf /var/lib/apt/lists/*

# Install Gazebo & Meshlab
RUN curl -sSL http://get.gazebosim.org | sh

# PX4 Install Script
RUN wget https://raw.githubusercontent.com/PX4/PX4-Autopilot/refs/heads/main/Tools/
   setup/ubuntu.sh && \
    wget https://raw.githubusercontent.com/PX4/PX4-Autopilot/refs/heads/main/Tools/
        setup/requirements.txt && \
    chmod +x ubuntu.sh && \
    ./ubuntu.sh && \
    rm ubuntu.sh && \
    rm requirements.txt

# Install MAVROS
RUN apt-get install -y ros-humble-mavros \
    ros-humble-mavros-extras && \
    wget https://raw.githubusercontent.com/mavlink/mavros/master/mavros/scripts/
        install_geographiclib_datasets.sh && \
    chmod +x install_geographiclib_datasets.sh && \
    ./install_geographiclib_datasets.sh && \
    rm install_geographiclib_datasets.sh

# Install Python packages
RUN pip install --upgrade pymavlink MAVProxy

# Edit .bashrc
RUN echo '#! /bin/bash' >> /home/${USER}/.bashrc && \
    echo 'source /opt/ros/humble/setup.bash' >> /home/${USER}/.bashrc

# Switch to the specified user
USER ${USER}
```

This Dockerfile sets up a non-root user, installs essential ROS and PX4 packages, configures the environment, and ensures that all dependencies are readily available. The '.bashrc' configuration automatically sources ROS upon startup, providing a ready-to-use ROS environment in the container [10], [11]. Additionally, the setup includes Gazebo installation to support simulation requirements [12], along with PX4 setup scripts [13].

The motor failure plugin was based on

References

[1] A. Freddi, S. Longhi, and A. Monteriù. “Actuator Fault Detection System for a Mini-Quadrotor”. In: *Journal of Intelligent Robotic Systems* 61.1 (2011), pp. 19–37.

[2] Ngoc Phi Nguyen and Sung Kyung Hong. “Sliding Mode Thau Observer for Actuator Fault Diagnosis of Quadcopter UAVs”. In: *International Journal of Control, Automation and Systems* 17.6 (2019), pp. 1484–1496.

[3] Zhaohui Cen et al. “Robust Fault Diagnosis for Quadrotor UAVs Using Adaptive Thau Observer”. In: *IEEE Transactions on Control Systems Technology* 28.4 (2020), pp. 1301–1310.

[4] Khalid Mohsin Ali and Alaa Abdulhady Jaber. “Comparing Dynamic Model and Flight Control of Plus and Cross Quadcopter Configurations”. In: *International Journal of Advanced Robotic Systems* 13.4 (2016), pp. 1–12.

[5] 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`.

[6] 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`.

[7] Sihao Sun et al. “Upset Recovery Control for Quadrotors Subjected to a Complete Rotor Failure from Large Initial Disturbances”. In: *IEEE Transactions on Control Systems Technology* 28.5 (2020), pp. 1968–1980. DOI: `10.1109/TCST.2020.2977470`.

[8] Vincenzo Lippiello, Fabio Ruggiero, and Diana Serra. “Emergency Landing for a Quadrotor in Case of a Propeller Failure: A PID-Based Approach”. In: *Journal of Intelligent & Robotic Systems* 67.1 (2012), pp. 37–53. DOI: `10.1007/s10846-012-9674-3`.

[9] 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`.

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

[11] PX4 Autopilot. *PX4 Autopilot Documentation.* 2024. URL: `https://docs.px4.io/master/en/`.

[12] Gazebo Simulator. *Gazebo Installation.* 2024. URL: `http://get.gazebosim.org`.

[13] PX4 Autopilot. *PX4 Autopilot Setup Script.* 2024. URL: `https://raw.githubusercontent.com/PX4/PX4-Autopilot/refs/heads/main/Tools/setup/ubuntu.sh`.