Inter IIT Tech Meet 13.0

Position and Altitude Control of Quadrotor with Single Motor Failure

Midterm 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. 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 enable safe landing along with custom trajectory following.

2 Motor Failure Detection

The study began with a thorough literature review on motor failure isolation. Neural network techniques were combined with a classical dynamics approach to identify the failed motor.

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 as shown in Figure 2.

2.1.1 Literature Review

- A. Freddi et al.'s work on this system is 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 algebraic differences of **estimated** state variables and **measured** state variables. In normal operation, the residuals stay near zero. When a fault occurs, they rapidly increase, aiding in the isolation of actuator faults.
- 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 quadcopters. It introduces the **Adaptive 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 detects motor faults, estimates their magnitude, tracks progression, and rejects external disturbances.
- Another reference 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 unmodeled dynamics by cascading more optimized observers with the ATO.

Note

One more paper found relevant was by Khalid Mohsin Ali et al. in [4]. To maintain uniformity across our work and **PX4** standards, we have used **CROSS** dynamics from this paper.

2.1.2 Setting up the ROS node

Software-In-The-Loop (SITL) simulation with PX4 was used, while interacting within Gazebo Classic's virtual world. This enabled PX4 to mimic real-world sensor readings and responses to control inputs. MAVROS, which acts as a bridge between ROS and PX4 was used to gather the live data of each drone flight. We also used **eProsima** `uXRCE-DDS` agent outside of PX4 to expose more topics under namespace `/fmu`. The topics subscribed are mentioned in the table above. The

Topic Name	Message Type	Description
`/mavros/imu/data`	`sensor_msgs/Imu`	Linear Acceleration,Angular Velocity,Orientation
`/motor_failure/motor_number`	`std_msgs/Int32`	Motor failure data
`/fmu/out/actuator_motors`	`px4_msgs/ActuatorMotors`	Motor Thrust Setpoint Data

Table 1: ROS Topics with Message Types and Data Descriptions

`/motor_failure/motor_number/` topic indicates the failure of any motor. This is defined by a custom plugin, the specifics of which are provided in the Workspace Setup subsection.

2.1.3 System Description

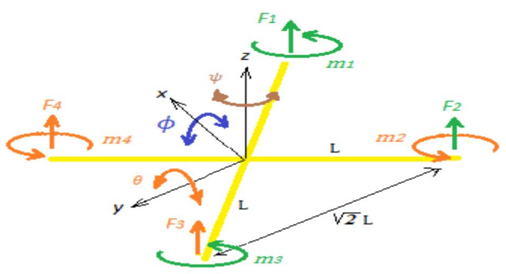


Figure 1: Schematic of the geometric configuration of the quadcopter

The control variables include:

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

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

$$\left\{\dot{x}(t) = Ax(t) + Bu(t) + h(x,u); \quad y(t) = Cx(t)\right. \tag{3}$$

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

2.1.4 Setting up the Thau's Observer

A stable observer for the system has the form

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

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 $\dot{\hat{x}}(t)$:

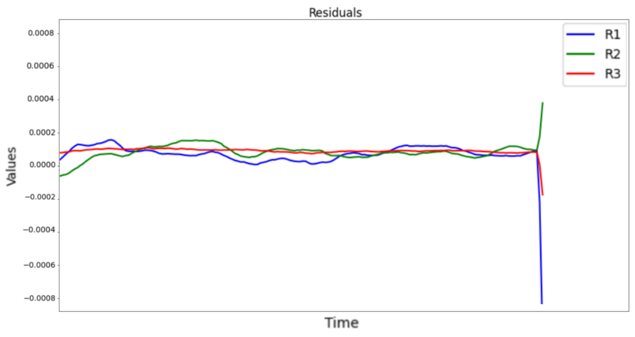
$$\begin{aligned} k_1 &= T_s \cdot \dot{x} \\ k_2 &= T_s \cdot \left(\dot{x} + \tfrac{1}{2}k_2\right) \end{aligned}$$

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

2.1.5 Calculating the Residuals

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

2.1.6 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** with **100%** accuracy with an average detection latency of **80ms**. Improving accuracy of detecting failures during agile flights is being worked on.

2.2 Machine Learning based Approach

2.2.1 Literature Review

- Several studies that address single motor failure detection in Unmanned Aerial Vehicles (UAVs) were reviewed. 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. The 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, a sliding window of 20 timestamps was created, 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 the detection approach, making it unsuitable for our requirements. Consequently, this approach was not adopted.
- Zhang et al. present a study on fault detection and identification (FDI) for quadcopters in [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**.However, this method is designed to pre-detect any faults in the quadrotor propellors and not to detect a motor failure in real time, trying to do which introduces significant latency in the predictions.

2.2.2 Our Approach

After reviewing various approaches in the literature, a **lightweight Recurrent Neural Network (RNN)** model was implemented for motor failure detection. Figure 3 illustrates the model framework, consisting of an input layer, two hidden layers, and an output layer. Predictions are made by **applying a threshold** to the output layer's neuron values.

Note

The model is designed to be highly 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.

2.2.3 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 2.1.2.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.

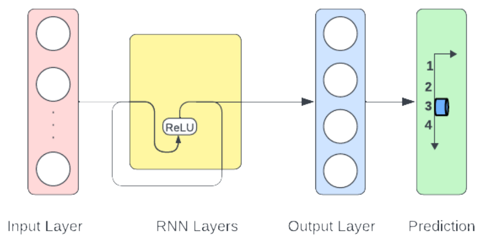


Figure 3: Model Layers

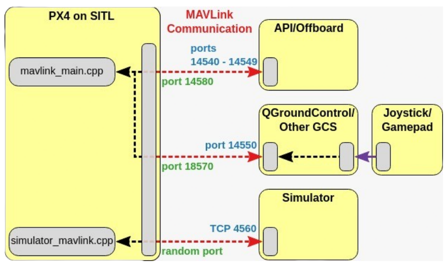


Figure 4: SITL Simulation

2.2.4 Training and Testing

Each row of the CSV file, an input vector of shape $(1, 10)$ from IMU sensor features, is paired with a label for the failed motor at that timestamp. The model processes each input sequentially, updating its memory to learn temporal patterns indicative of motor failure, achieving high accuracy in real-time 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.

Note

The model achieved an **accuracy of 100%** in detecting the motor failure with **0 false positives** in all of the 20 test flights. On average, as demonstrated in Figure 5, 9 sequential readings from the IMU sensor were taken by the model, which at 150hz refresh rate, converts to an **inference time of 180ms**.

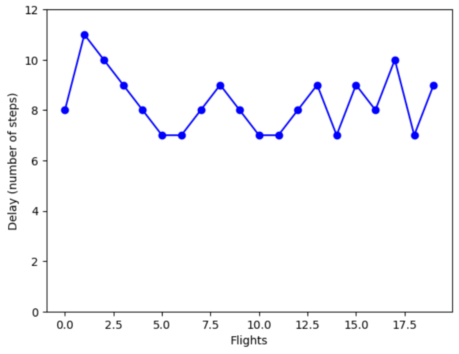


Figure 5: Latency across test flights

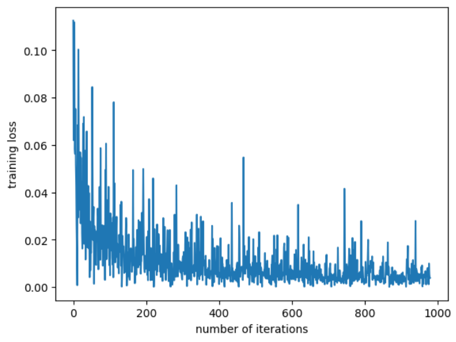


Figure 6: Training Loss vs Iterations

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

The study *Upset Recovery Control for Quadrotors Subjected to a Complete Rotor Failure from Large Initial Disturbances* by Sun et al. [7] was consulted, 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.

INDI incrementally adjusts control inputs in response to detected faults, providing gradual, adaptive compensation that maintains stability over time. NTSMC, by contrast, leverages terminal sliding surfaces to drive the system state to a target within a finite timeframe, offering strong robustness against sudden, large faults.

While INDI is effective for handling larger faults with steady corrections, it can struggle with smaller disturbances, affecting precision. NTSMC, though robust against large disturbances, may overcompensate when faced with minor faults due to its more aggressive control nature.

3.1.2 Emergency Landing for a Quadrotor using PID based approach

The work by Lippiello et al. on *Emergency Landing for a Quadrotor in Case of a Propeller Failure: A PID-Based Approach* [8], was referred to. It consisted of a **PID-based approach** to find solution for motor failure in a quadrotor during flight. 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 on 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.

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, a **Nonlinear Model Predictive Controller (NMPC)** that utilizes a custom-defined cost function, developed by Fang Nan in [9] was integrated .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.

The implementation of NMPC demonstrated promising results, even under extreme conditions where traditional controllers tend to struggle with stability. With this approach, return to the pre-failure reference position is prioritized, followed by 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.

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 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

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

$$\dot{\mathbf{p}} = \mathbf{v}, \tag{7}$$

$$\dot{\mathbf{q}} = \frac{1}{2}\mathbf{q} \circ \begin{bmatrix} 0 \\ \boldsymbol{\omega} \end{bmatrix}, \tag{8}$$

$$\dot{\mathbf{v}} = \mathbf{q} \otimes \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix} - \mathbf{g}, \tag{9}$$

$$\dot{\boldsymbol{\omega}} = \mathbf{I}^{-1}\left(\boldsymbol{\tau} - \boldsymbol{\omega} \times (\mathbf{I}\boldsymbol{\omega}) + \boldsymbol{\tau}_{\text{ext}}\right), \tag{10}$$

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.

3.2.3 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:

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

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:

$$\mathbf{x}_{i+1} = f(\mathbf{x}_i, \mathbf{u}_i), \quad i = k, \ldots, k+N-1, \tag{11}$$

  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:

$$\mathbf{u}_{\min} \leq \mathbf{u} \leq \mathbf{u}_{\max}, \tag{12}$$

  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 $\mathbf{Q}$, $\mathbf{Q}_N$, and $\mathbf{R}$ are positive-definite weight matrices, which were optimally obtain as follows :
  - $\mathbf{Q} = \text{diag}(8, 8, 80, 6, 0, 0.1, 0.1, 0.1, 5, 5, 0.1, 0.1, 0.1, 0.1)$
  - $\mathbf{Q}_N = \text{diag}(8, 8, 80, 6, 0, 0.1, 0.1, 0.1, 5, 5, 0.1)$
  - $\mathbf{R} = \text{diag}(0.1, 0.1, 0.1, 0.1)$

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.

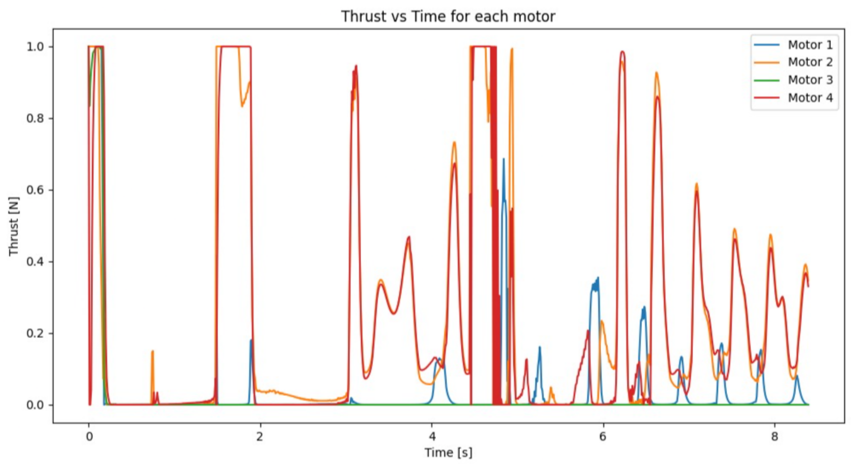


Figure 7: Thrust V/s Time for Each Motor before and after Motor Failure

3.2.4 Attitude Error and Cost Vector

We compute the error which and the cost vector $\mathbf{iy}_\mathrm{i}$ 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 $\mathbf{q}_\mathrm{ref}$ 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.

$$\mathbf{q_e} = \mathbf{q_{ref}} \circ \mathbf{q}^{-1} \tag{13}$$

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

  Note

  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 the following operation.

$$\mathbf{q_{xy}} = \mathbf{q_e} \circ \mathbf{q_z}^{-1} \tag{14}$$

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

3.2.5 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.

3.2.6 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 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.

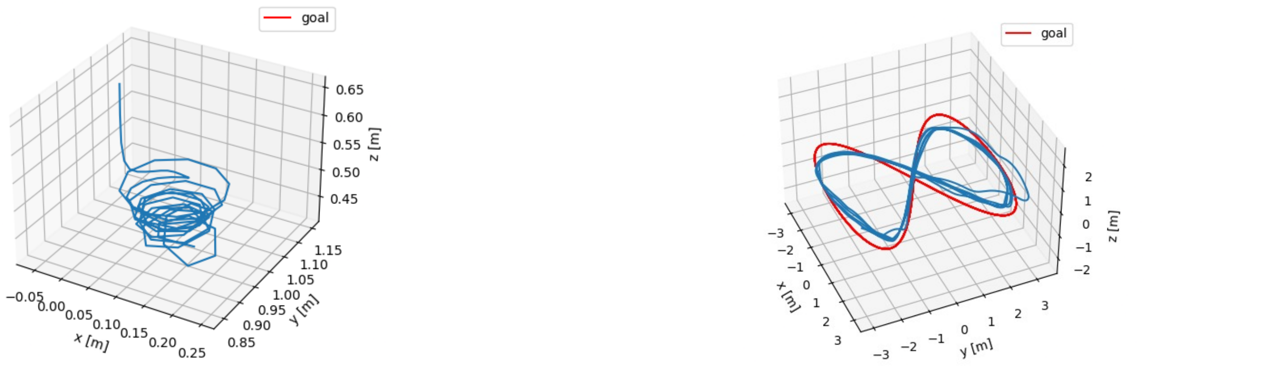
Note

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

3.2.7 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.

- **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]

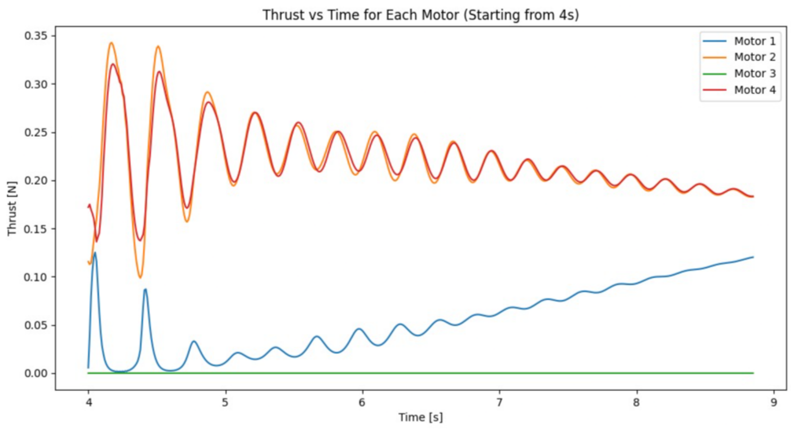
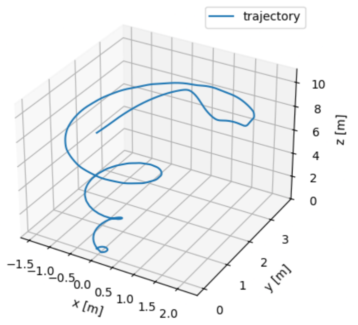
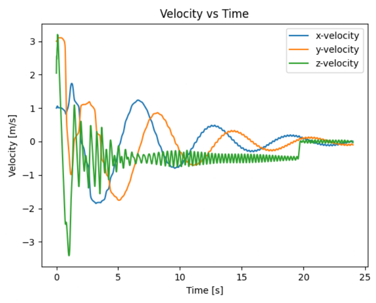


Figure 9: Thrust for each motor while stable hover



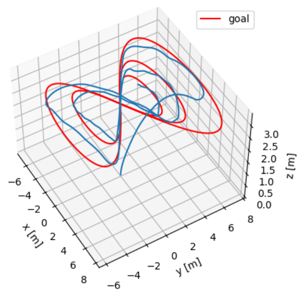
(a) Trajectory tracked during emergency landing



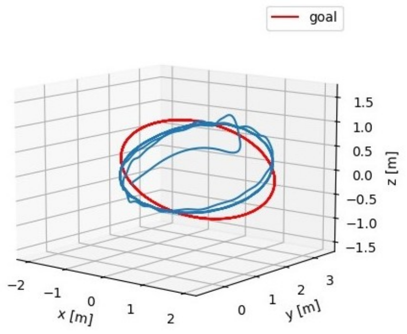
(b) Velocity in x, y, z during controlled landing

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

4 Challenges Ahead and Next Steps

Issues with Controller:

- The reference angular velocity $\mathbf{w}_{\text{ref}}$ after motor failure is manually set, but we plan to dynamically adjust it based on altitude changes.
- The solver struggles with distant goals due to high values, so we divide the target into smaller intermediate goals.
- Drag is preliminarily implemented and needs further testing in Gazebo for accuracy and parameter adjustments.
- To ensure a smooth transition between the pre- and post-failure algorithms following motor failure detection, we plan to incorporate a gradual reduction in the maximum motor limits, rather than an abrupt drop to zero. This approach aims to prevent solver errors and improve system stability.

Issues with Classical Observer Failure Detection:

- There is some unmodelled parameters left from observer which require an **Adaptive Thau Observer (ATO)** cascaded with more non-linear observers to take care of unmodelled dynamics and noise filtering.
- Robustness of the current observer is being worked on. We are yet to manage cases of highly agile flights.
- We can further improve latency from **80ms** via **ATO**

Issues with ML-Based Failure Detection:

- The model runs slower on PX4 controller, likely due to timestamp mismatches between CSV data & real-time inputs.
- Data generation couldn't be automated in ROS2; Gazebo simulation failed, requiring manual waypoint input via QGC. This step would be automated via running a script and forming dataset in formatted way using data from MAVROS
- The model gives random values at takeoff due to improper handling of file transitions during training, affecting initial predictions. This will be fixed by creating new training dataset in which the quadrotor does not takes off for a couple of seconds so that the output neuronal values from the previous files vanishes.

Steps Ahead:

- Motor failure detection has been integrated inside the PX4 firmware for low latency. The same is left to be done for the implemented NMPC controller.
- Further step would be testing our simulation results on real hardware to verify functionality and ease of use, though this poses challenges due to a non-ideal environment and the risk of hardware damage.
- 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 utilized Docker to set up a consistent workspace across Linux versions, eliminating code-based dependencies. All dependencies are documented in the code repository. The Dockerfile creates a non-root user, installs essential ROS and PX4 packages, configures the environment, and ensures all dependencies are accessible. The `.bashrc` file auto-sources ROS and necessary packages on startup, providing a ready-to-use ROS environment [10] [11]. Gazebo, with PX4 setup scripts supports simulation requirements, using the IRIS model in Gazebo Classic.[12].

The motor failure plugin [13] was made compatible with ROS2. The code transition from ROS 1 to ROS 2 involves switching from `ros::SubscribeOptions` to `create_subscription` and updating the message type to `std_msgs::msg::Int32` with the `msg::` namespace. Additionally, callback binding now uses `std::function` and `SharedPtr` for automatic memory management, replacing raw pointers and manual binding. IRIS model was modified to include Gazebo motor failure plugin in the SDF file, as detailed in the documentation, with the necessary configuration for motor failure topics and plugin integration.

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] 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. DOI: `10.5937/fme2204683M`.

[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: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020, pp. 4273–4279. DOI: `10.1109/ICRA40945.2020.9197239`.

[8] 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`.

[9] 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`.

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

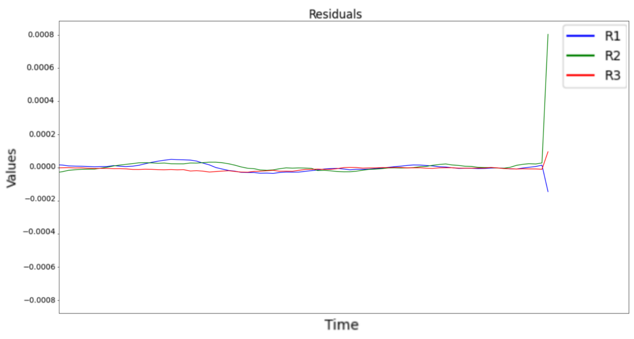
[11] PX4 Autopilot. *PX4 Autopilot Documentation*. 2024. URL: `https://docs.px4.io/main/en/index.html`.

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

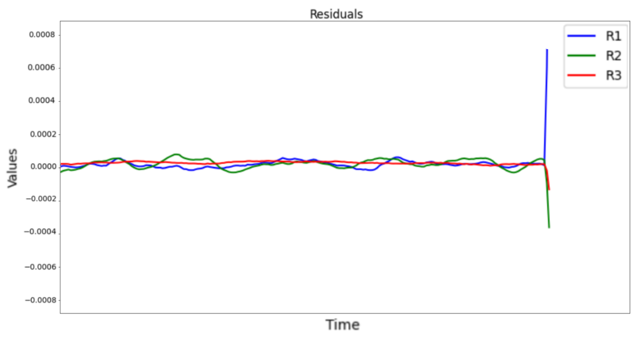
[13] PX4 Autopilot Developers. *PX4-SITL_gazebo-classic: gazebo_motor_failure_plugin.cpp*. `https://github.com/PX4/PX4-SITL_gazebo-classic/blob/f9bc7e34a754b6f204a8b3825c2b132587574cba/src/gazebo_motor_failure_plugin.cpp`. 2024.

6 Appendix

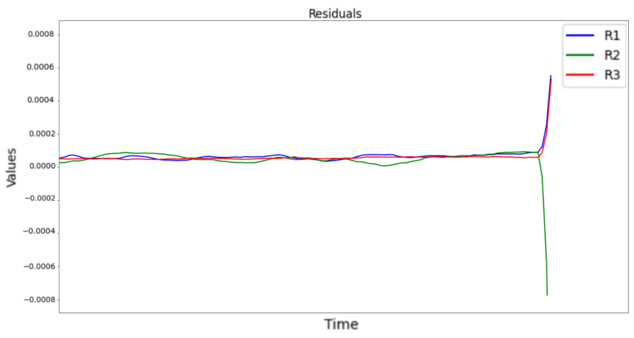
6.1 Residuals with Classical Observer based detection



(a) Residuals when Motor 2 is failed



(b) Residuals when Motor 3 is failed



(c) Residuals when Motor 4 is failed