ideaForge Aerial Robotics

Inter IIT Tech Meet 13.0

Final Submission Report

Author:

Team 67

Contents

I. Introduction

Unmanned Aerial Vehicles (UAVs), particularly quadrotors, are integral to modern surveillance, mapping, and reconnaissance missions across diverse sectors, from defense to public safety. However, a fundamental vulnerability of quadrotor UAVs is the potential for catastrophic instability in the event of a single motor failure. This report presents our strategic analysis and innovative solutions that we have formulated to address the challenges articulated in ideaForge's problem statement. Our insights and solutions aim to empower ideaForge with a forward-thinking approach to enhance the reliability of their quadcopter systems.

II. Problem Understanding

1) **Implicit Motor Failure Detection:** Identify when a motor fails without explicit sensors.
2) **Controlled Landing:** Adjust control of the remaining three motors to enable a stable, spinning descent for an immediate landing.
3) **Stable Hovering:** Modify controls to hold altitude and position with three motors, achieving a stable hover.
4) **Navigation and Return Control:** Enable the UAV to perform basic navigation and altitude adjustments, allowing it to land safely to a landing point.

Each solution stage aims to enhance the quadrotor's resilience to single motor failure, gradually increasing its operational capability despite the compromised control system.

III. Motor Failure Detection

Among the two proposed methods for motor failure detection are a threshold-based approach and a machine learning algorithm, specifically a MLP based approach. The threshold-based approach utilizes sensor data (gyroscope, accelerometer, magnetometer) to identify anomalies, while the Multi-Layer Perceptron (MLP) model learns patterns from historical data to detect failures.

A. *Threshold Approach for Motor Fault Detection*

The solution implements a real-time drone fault detection system that monitors sensor data (IMU) for anomalies, compares them to thresholds, and logs detected faults.

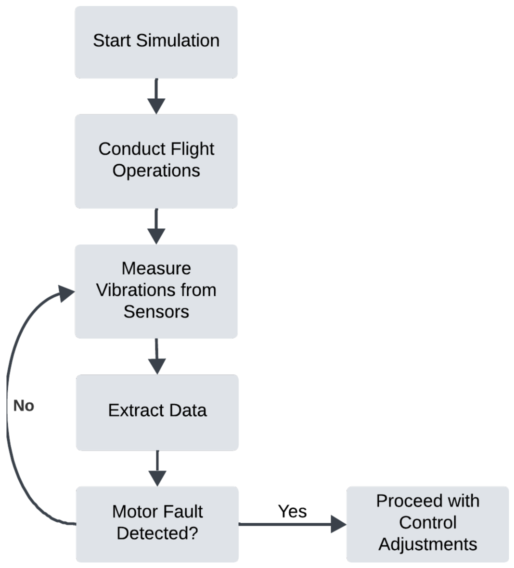
Start Simulation
Conduct Flight Operations
Measure Vibrations from Sensors
Extract Data
Motor Fault Detected?
No
Yes
Proceed with Control Adjustments

Fig. 1: Failure Detection Flowchart

Thresholds were established for fault detection in various sensor parameters. Data is collected from a simulated quadrotor(IRIS) in Gazebo, and patterns in these measurements are analyzed during various motor failure scenarios. Threshold values are established for detection, and the concept of sensitivity is introduced to balance detection latency and noise resilience, ensuring robust and practical failure detection.

*1) **An overview of the approach**[1]:* The above proposed method involves simulating quadrotor flights under different motor failure conditions, collecting IMU data, plotting the readings, and determining thresholds for failure detection. Sensitivity is optimized to balance detection accuracy and response time.

i) **Simulation Setup:**
- Environment: Gazebo Simulator.
- Quadrotor Model: IRIS drone.
- Sensors Used:
  - Gyroscope for measuring angular velocities g_x, g_y, g_z .
  - Accelerometer for measuring linear accelerations a_x, a_y, a_z.
  - Magnetometer for measuring magnetic field components m_x, m_y, m_z

ii) **Simulation Scenarios:**
- Normal Conditions: Simulation without any noise.

- Noise Conditions: Wind condition with variance was introduced with the following parameter(in m/s) :
  - Mean Wind Velocity = 4.0
  - Maximum Wind Velocity = 20.0
  - Variance in Wind Velocity = 1
  - Mean Wind Direction x,y,z = [0 1 0]
  - Variance in Wind Direction = 0.2

*2) **Data Collection**:* To ensure the robustness and reliability of the motor failure detection method, all possible scenarios have been considered, both under normal conditions and with noise. These scenarios include:

i) **Hovering Scenarios:**
- Normal Conditions:
  - Hover $\rightarrow$ Motor 1 Failure
  - Hover $\rightarrow$ Motor 2 Failure
  - Hover $\rightarrow$ Motor 3 Failure
  - Hover $\rightarrow$ Motor 4 Failure
- With Noise:
  - Hover $\rightarrow$ Motor 1 Failure with Noise
  - Hover $\rightarrow$ Motor 2 Failure with Noise
  - Hover $\rightarrow$ Motor 3 Failure with Noise
  - Hover $\rightarrow$ Motor 4 Failure with Noise

ii) **Directional Scenarios:** Maximum possible roll, pitch, and yaw were set as how an ideal UAV operates (30° roll; 30° pitch; π rad/sec yaw clockwise and counterclockwise)
- Normal Conditions:
  - Takeoff $\rightarrow$ Roll Axis $\rightarrow$ Motor Failure
  - Takeoff $\rightarrow$ Pitch Axis $\rightarrow$ Motor Failure
  - Takeoff $\rightarrow$ Yaw Axis Clockwise $\rightarrow$ Motor Failure
  - Takeoff $\rightarrow$ Yaw Axis Counter-Clockwise $\rightarrow$ Motor Failure
- With Noise:
  - Takeoff $\rightarrow$ Roll Axis $\rightarrow$ Motor Failure with noise
  - Takeoff $\rightarrow$ Pitch Axis $\rightarrow$ Motor Failure with noise
  - Takeoff $\rightarrow$ Yaw Axis Clockwise $\rightarrow$ Motor Failure with noise
  - Takeoff $\rightarrow$ Yaw Axis Counter-Clockwise $\rightarrow$ Motor Failure with noise
- Parameters Recorded:
  For each test case, the following IMU data is collected at a frequency of 200 hertz:
  - **Gyroscope:** Angular velocities g_x, g_y, g_z in rad/s.
  - **Accelerometer:** Linear accelerations a_x, a_y, a_z in m/s².
  - **Magnetometer:** Magnetic field components m_x, m_y, m_z in gauss.
- Noise Scenarios: Data was also collected and thresholds were set under noisy conditions to test robustness.

*3) **Data Visualization**:* Data from the IMU sensors is visualized using Python's matplotlib library. For each test case:

i) **Gyroscope Data:** Plots of angular velocities g_x, g_y, g_z against time.
ii) **Accelerometer Data:** Plots of linear accelerations a_x, a_y, a_z against time.
iii) **Magnetometer Data:** Plots of magnetic field components m_x, m_y, m_z against time

*4) **Observations**:* The motor failure analysis was conducted based on variations in sensor data. The observations were categorized into three phases: pre-failure, during failure, and post-failure, with specific trends noted in each phase.

i) **Pre-Failure:**
- *Gyroscope:* the gyroscope readings indicated stable angular velocity near 0 rad/s across all axes.
- *Accelerometer*: Data showed nominal readings consistent with stable flight conditions, indicating no significant vibrations or anomalies.
- *Magnetometer*: Readings confirmed a consistent orientation without any abrupt variations, aligning with the stable flight behavior.

ii) **During Failure:** The motor failure triggered distinct changes in the sensor data, particularly in the gyroscope, accelerometer, and magnetometer readings:
- *Gyroscope*:
  - Sudden spikes or drops in angular velocity were observed in one or more axes.
  - Notable oscillations in g_x and g_y were detected immediately post-failure, high-

lighting instability.
  - A significant spike or drop in g_z was recorded, depending on the nature and type of motor failure.
- *Accelerometer*: Marked variations in acceleration values indicated increased vibrations and instability caused by the failure.
- *Magnetometer*: Abrupt *shifts* in orientation readings were observed, further confirming the occurrence of motor failure and its impact on the drone's stability.

iii) **Post-Failure:** After the initial failure response, the system exhibited signs of stabilization over time. But, the received data was due to vibrations caused by the motors even after the failure occurred as they were not killed at once

5) *Sensitivity and its Role:*

i) **Sensitivity:** Sensitivity refers to how responsive the failure detection method is to changes in sensor data:
- High Sensitivity: Faster detection but prone to false positives due to noise.
- Low Sensitivity: Reduced false positives but higher detection latency.

A balanced sensitivity is selected by analyzing IMU data under normal and noisy conditions to avoid false positives while ensuring practical detection latency.

ii) **Sensitivity Trade-Off:** From the sensor data, a balance is required to ensure:
- No false positives: There should be no false positives to avoid noise-induced failure detection.
- Practical latency: Ensure timely detection for quadrotor control.

The sensitivity level is determined by:
- Analyzing noise amplitude in the graphs.
- Setting thresholds to exclude noise-induced fluctuations while ensuring early failure detection.

iii) **Sensitivity Optimization:** Using the plotted data:
- Thresholds are adjusted based on noise levels and failure onset patterns.
- These thresholds balance sensitivity and robustness, ensuring timely detection without false positives.

iv) **Results:**
- *Accuracy*:
  - Balanced sensitivity detected motor failures in all simulated scenarios.
  - Minimal false positives were observed under the optimized thresholds.
- *Performance Under Noise:* The noise introduces minor fluctuations in sensor data, however the overall pattern of these fluctuations remains consistent with those observed in an ideal environment. As a result, the detection method remains robust, and noise has minimal impact on the accuracy of motor failure detection.
- *Latency*: The latency of approximately **0.4835–0.6323 seconds** was measured using MAVSDK with Python code, which connects to PX4 via the MAVLink protocol. However, since our implementation is directly embedded in PX4 using C++, the actual latency is expected to be significantly lower, as the overhead introduced by MAVLink communication and external scripting is eliminated. This **direct integration** optimizes the system's performance by reducing communication delays and leveraging the efficiency of C++ for real-time operations.

B. *Data Collection for Machine Learning Approaches*

Next, we adopted a data-driven approach for the machine learning model that leverages accelerometer, gyroscope, and magnetometer data to identify anomalies indicative of motor failure. In this context, we classified motor states into five categories: Safe Operation, Failure in Motor 1, Failure in Motor 2, Failure in Motor 3, and Failure in Motor 4. The process for developing the dataset and selecting relevant parameters involved several key steps:

*1) **Data Collection**:* The dataset was generated by recording sensor data in CSV files during controlled experiments under various flight conditions with induced motor failures. The experiments were designed to capture comprehensive sensor readings for multiple failure scenarios. The quadrotor's roll and pitch were limited to a maximum of ±30°, and yaw data was recorded at a rate of up to 180°/sec to ensure realistic flight dynamics.

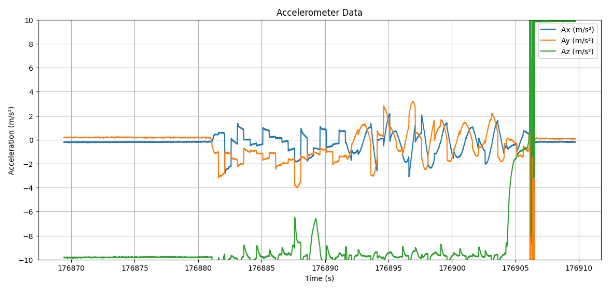

Fig. 2: Accelerometer Reading

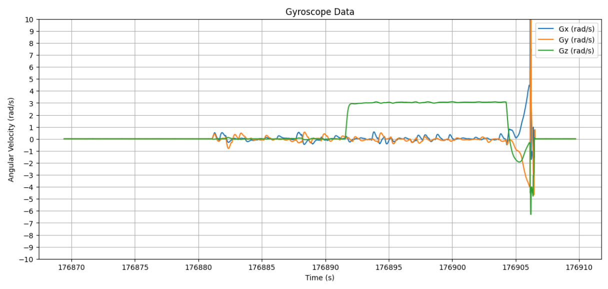

Fig. 3: Gyroscope Reading

For each motor failure, the following test scenarios were performed:

- Pitch Angles: $\pm 10°$, $\pm 20°$, $\pm 30°$.
- Roll Angles: $\pm 10°$, $\pm 20°$, $\pm 30°$.
- Yaw Rotations: Clockwise and Counterclockwise.

Each scenario was repeated across **five test cases**, resulting in a total of 75 datasets per motor failure.

The features used in the dataset include: Linear accelerations a_x, a_y, a_z (along the x, y, and z axes), Angular velocities g_x, g_y, g_z (around the x, y, and z axes), magnetic field reading m_x, m_y, m_z (along the x, y, and z axes) for multiple failure events.

Enhancements for Robustness: Sensor noise and random disturbances were artificially introduced to each dataset to mimic real-world conditions and reduce false positives during failure detection. A timestamp column was included to map temporal dependencies, though it was excluded during training to prevent bias in the ML models.

This structured data collection enabled us to build a comprehensive dataset capturing various sensor readings corresponding to each type of motor failure.

*2) **Pre-processing**:* Data was scaled using $StandardScaler$ to ensure uniform weighting. The example transformation is

$$z = \frac{x - \mu}{\sigma} \tag{1}$$

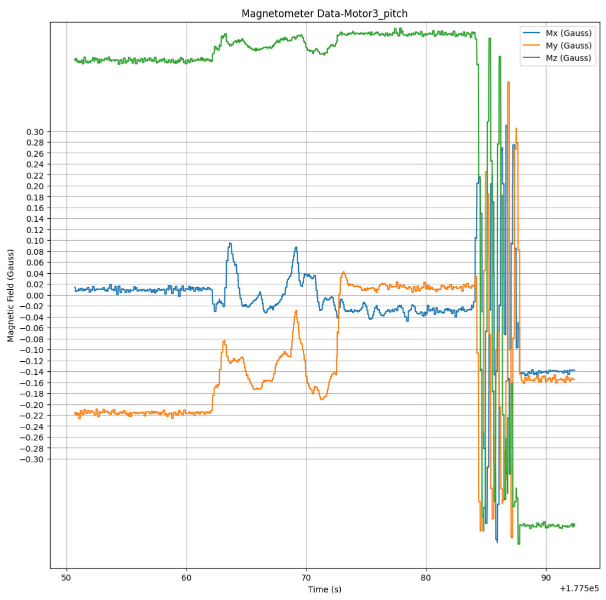

Fig. 4: Magnetometer Readings

where x is the raw value, μ is the mean, and σ is the standard deviation.

*3) **Data Visualization and Analysis**:* We analysed and visualized the sensor data, plotting $a_x, a_y, a_z, g_x, g_y, g_z, m_x, m_y$ and m_z values. Distinct variations in these parameters were observed for different motor failures, providing initial insights into how sensor data patterns might reveal specific motor anomalies.

*4) **Parameter Selection**:* Various combinations of accelerometer (a_x, a_y, a_z) gyroscope (g_x, g_y, g_z) and magnetometer (m_x, m_y, m_z) parameters were tested to identify the optimal set that maximizes detection accuracy. This process involved iterative testing to ensure that the selected parameters reliably distinguish between different failure states.

*C. **Multi-Layer Perceptron (MLP) Model***

[2] A Multi-Layer Perceptron (MLP) is an artificial neural network commonly used for supervised learning tasks. MLP consists of three main layers:

i) Input Layer: Receives input data with various features.
ii) Hidden Layers: Comprise a series of interconnected nodes (or neurons) that learn patterns in the data.
iii) Output Layer: Produces the final prediction, often in probabilities or classes.

*1) **Our Approach**:* An MLP is trained on labeled data, adjusting the weights and biases of the connections to minimize the error in its predictions. This learning process, called backpropagation, enables the model to capture complex patterns and relationships within the data, making it highly suitable for classification tasks. It is suitable for this project because of its ability to model non-linear relationships inherent in UAV dynamics, flexibility in capturing subtle variations in sensor data indicative of failures, and applicability for supervised classification tasks.

The MLP was trained on labeled data representing the five motor states in our motor failure detection system. This model demonstrated a high level of accuracy and reliability in detecting failures. Specifically, the MLP's **benefits** include:

i) **Accurate Classification:** The MLP consistently identifies the motor involved in a failure.
ii) **Real-Time Detection:** It meets real-time motor failure detection requirements in ROS (Robot Operating System), which is essential for maintaining stability and safety.
iii) **Robustness in Hovering Conditions:** The MLP can distinguish between normal and failed motor conditions even under minor disturbances, making it a suitable choice for real-time failure classification.

*2) **The dataset**:* The dataset comprises accelerometer and gyroscope readings $(a_x, a_y, a_z, g_x, g_y, g_z)$ captured during controlled experiments where motor failures were induced under various pitch, roll, and yaw conditions at angles of 10°, 20°, and 30°.

*3) **MLP Architecture**:*

- **Input Layer:** 6 neurons corresponding to the features$(a_x, a_y, a_z, g_x, g_y, g_z)$.
- **Hidden Layers:** There are two fully connected layers with 128 and 64 neurons, respectively and the Activation function: $ReLU$ (Rectified Linear Unit) ensures non-linearity and efficient training, is defined as:

$$f(x) = \max\{0, x\} \tag{2}$$

- **Output Layer:** 5 neurons corresponding to the classes (Normal, Motor 1, Motor 2, Motor 3, Motor 4). Activation function: Softmax to output probabilities is defined as:

$$P(y_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

- **Loss Function:** Categorical Cross-Entropy for multi-class classification is defined as :

$$L = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

*4) **Results**:*

- **Performance Metrics:**
  - **Accuracy:** 0.84
  - **Precision:** 0.84
  - **Recall:** 0.84

  Classification Report:

Class	Precision	Recall	F1-Score	Support
0	0.99	1.00	0.99	13269
1	0.85	0.85	0.85	3988
2	0.88	0.86	0.87	3909
3	0.81	0.78	0.80	3950
4	0.63	0.35	0.45	4025
Accuracy			0.84	29141
Macro Avg	0.78	0.77	0.76	29141
Weighted Avg	0.84	0.84	0.83	29141

- **Confusion Matrix:**

$$\begin{bmatrix} 13220 & 10 & 24 & 8 & 7 \\ 33 & 3408 & 187 & 233 & 127 \\ 49 & 225 & 3348 & 183 & 104 \\ 16 & 154 & 107 & 3076 & 597 \\ 39 & 192 & 154 & 2238 & 1402 \end{bmatrix}$$

D. Integration of failure detection algorithm into PX4 Autopilot

The multi-layer Perceptron (MLP) model has been integrated into the PX4 firmware to enable real-time motor failure detection. The trained model processes sensor data from the UAV's onboard sensors classifies the operational state and triggers appropriate actions when a failure is detected.

*1) **Integration Workflow**:*

- **Communication Between ORB Topic and Control Allocator:** The ControlAllocator module has been enhanced to subscribe to essential sensor topics, specifically accelerometer, gyroscope, and magnetometer. Dedicated subscribers added within the

ControlAllocator.hpp file to fetch real-time sensor data at **200 Hz** from ORB topics, providing high-resolution inputs for motor fault detection and control allocation.

- **Mode Inference and Motor Failure Logic:** The motor failure detection is implemented within *ControlAllocator.cpp* file, leveraging the subscribed sensor data at **200 Hz** to identify anomalies in motor behaviour. The fault detection mechanism is integrated into the mode inference logic, ensuring that motor failures are detected and addressed in real time, thereby improving the reliability and robustness of the overall system.

2) ***PX4 Customization:*** To adapt the PX4 Autopilot for our project, several customizations were implemented to integrate our algorithm for motor failure detection and recovery. These modifications aim to bypass the default failsafe mechanism and optimize the system for better performance under failure conditions.

i) **Disabling Default Failsafe Mechanisms:** The PX4-Autopilot framework includes built-in motor failure detection mechanisms that activate failsafe actions when a motor failure is identified. These actions typically generate failure messages and transition the drone into a failsafe mode to prioritize safety. To integrate our custom motor failure detection and recovery algorithm, these default mechanisms were selectively disabled by commenting out the code responsible for triggering failsafe actions. The modifications were applied in the following files:
   - ***FailureDetector.cpp:*** Commented out lines responsible for detecting motor failures and triggering failsafe actions.
   - ***Framework.cpp:*** Disabled code handling automatic failsafe transitions upon motor failure detection.
   - ***FailureDetectorCheck.cpp:*** Removed logic related to built-in motor failure handling during system checks.
   - ***Health and Arming Checks Folder:*** Edited files in this folder to bypass default failure messages and arming restrictions caused by motor failure detection.

   By disabling these components, the system was configured to allow the implementation of our custom algorithm post-failure detection, without the interference of PX4's default failsafe mechanisms.

ii) **Optimizing Drone Mass for Enhanced Thrust:** The default drone mass in the PX4 configuration was initially set to 1.5 kg, which proved excessive for the thrust capabilities of the remaining motors in the event of a motor failure. To address this, the drone's mass was reduced to **1.35 kg** in the configuration files. This strategic adjustment ensured that the remaining motors could produce the enough thrust efficiently.

iii) **Enabling Custom Algorithm Integration:** The primary objective of these modifications was to facilitate the integration of a custom motor failure detection and handling algorithm. By disabling PX4's default failsafe mechanisms, the system is now capable to operate independently, allowing it to respond to failures according to our tailored logic.

IV. Fault-Tolerant Control Algorithm

[3]The proposed fault-tolerant control algorithm ensures a quadrotor can recover and maintain stability during a single motor failure by utilizing Nonlinear Model Predictive Control (NMPC) and Reinforcement Learning (RL). NMPC optimizes control inputs in real-time by minimizing a cost function over a prediction horizon, ensuring adaptive stability and control. RL, using algorithms such as Advantage Actor-Critic (A2C) and Proximal Policy Optimization (PPO) in simulations, enables the system to learn adaptive control policies through iterative interactions with its environment. This combination of NMPC and RL provides a robust framework for addressing motor failure, enhancing the quadrotor's reliability and resilience.

A. NMPC: Core Concepts and Assumptions

The Fault-Tolerant Nonlinear Model Predictive Control (NMPC) algorithm is designed for quadrotor control. The NMPC optimizes control inputs at each time step, minimizing a cost function over a prediction horizon while accounting for system dynamics, constraints, and potential faults. Key concepts include:

- **Nonlinear Model Predictive Control (NMPC)**: Uses a mathematical model of the

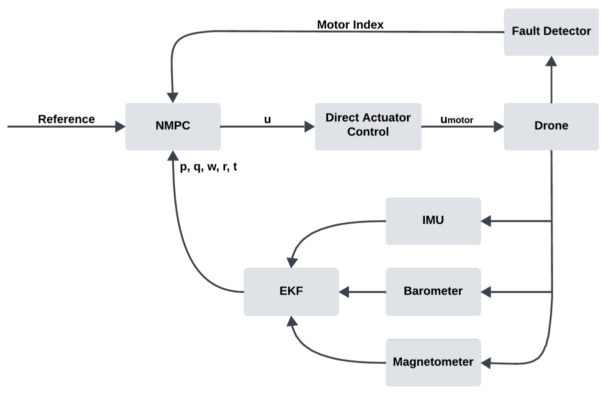

Fig. 5: NMPC flow chart

quadrotor's nonlinear dynamics to predict future states over a specified horizon. It calculates optimal control inputs (e.g., rotor thrusts) at each step, applying only the first input and continuously recalculating in a receding horizon manner, ensuring dynamic limits are respected.

- **Fault Tolerance**: NMPC compensates for partial or complete failures of specific actuators (e.g., motors) by redistributing control efforts among functional actuators, allowing the quadrotor to maintain its trajectory despite faults.
- **Assumptions**: The NMPC algorithm is based on the following assumptions:
  - System Dynamics: Quadrotor's dynamics, including translational, rotational behaviour.
  - Fault Detection: Actuator faults (e.g., motor failures).
  - Limited Prediction Horizon: NMPC operates over a finite horizon.
  - Constraints on Inputs and States: Physical limits on inputs (thrust, torque) and states (position, velocity, attitude).

B. *Control Algorithm Design*

The control algorithm operates as follows:

- **State Definition**: The quadrotor's state vector $\mathbf{x}$ includes position $\mathbf{p}$, orientation $\mathbf{q}$ (as a quaternion), linear and angular velocities $\mathbf{v}$ and $\boldsymbol{\omega}$, and thrust $\mathbf{t}$. The input vector $\mathbf{u} = [u_1, u_2, u_3, u_4]^\top$ corresponds to the control inputs (thrust or motor speeds).
- **Dynamic Constraints**: The NMPC enforces the quadrotor dynamics at each prediction step. The discretized dynamics model described as:

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

  where $\mathbf{x}_i$ is the state at a time step $\mathbf{i}$, and $\mathbf{u}_i$ is the control input (thrust) at that time step. provides the prediction of the quadrotor's future states over the horizon based on the current state and control inputs.
- **Control Input Bounds and State Constraints:** The control input bounds ensure that the thrust commands are within the physical limits: $\underline{\mathbf{u}} \le \mathbf{u} \le \bar{\mathbf{u}}$ where $\underline{\mathbf{u}}$ is the lower bound (often zero), and $\bar{\mathbf{u}}$ is the upper bound, which limits the maximum thrust the motors can provide. State constraints enforce operational safety (e.g., position bounds or maximum angles)
- **Optimization Problem**[3]: The NMPC solves an optimization problem to minimize the cost function under these constraints. It generates an optimal control sequence $\mathbf{u}_{k:k+N-1}$ over the horizon, from which only the first input $\mathbf{u}_k$ is applied to the system. This optimization problem is solved at every time step in a receding horizon manner.

  $$\underset{\mathbf{u}_{k:k+N-1}}{min} \{\mathbf{y}_N^\top \mathbf{Q}_N \mathbf{y}_N + \sum_{i=k}^{k+N-1} \mathbf{y}_i^\top \mathbf{Q} \mathbf{y}_i + \mathbf{u}_i^\top \mathbf{R} \mathbf{u}_i\}$$

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

  $$\underline{\mathbf{u}} \le \mathbf{u} \le \bar{\mathbf{u}}$$

C. *Cost Function Structure in NMPC*

A cost function in control systems, such as NMPC, is a mathematical expression that quantifies system performance by penalizing deviations from desired behaviors. It assigns penalties to deviations from the reference trajectory (target position and orientation) and control effort, using weighting matrices Q and R. It typically consists of two main components:

i) **Running Cost:** The accumulated cost over each time step within the horizon, penalizing deviations from the desired state (e.g., position, velocity, attitude) and excessive control efforts (thrust inputs).
   - *State Error Cost:* $\mathbf{y}_i^\top \mathbf{Q} \mathbf{y}_i$
     The state error cost penalizes deviations from the reference trajectory at each step.

- *Control Effort Cost:* $\mathbf{u}_i^\top \mathbf{R} \mathbf{u}_i$
  The control effort cost penalizes large or abrupt control inputs, ensuring smooth and efficient motor commands.

ii) **Terminal Cost:** The terminal cost $\mathbf{y}_N^\top \mathbf{Q}_N \mathbf{y}_N$ penalizes deviations in the final state, ensuring the quadrotor achieves its desired state. The weight matrix $\mathbf{Q}_N$ prioritizes specific components of the final state, balancing stability and tracking accuracy.

D. Derivation of the NMPC Cost Function

The objective of the cost function is to minimize state deviations and control effort while adhering to system dynamics and input constraints, such as actuator limits.
The optimization problem is formulated as follows:

$$\underset{\mathbf{u}_{k:k+N-1}}{min} \{\mathbf{y}_N^\top \mathbf{Q}_N \mathbf{y}_N + \sum_{i=k}^{k+N-1} \mathbf{y}_i^\top \mathbf{Q} \mathbf{y}_i + \mathbf{u}_i^\top \mathbf{R} \mathbf{u}_i\}$$

Where:

i) $\mathbf{y}_i$ is the deviation of the system's state at time step i from the desired reference state
ii) $\mathbf{Q}$ and $\mathbf{Q}_N$ are positive-definite weight matrices applied to penalize state deviations, with $\mathbf{Q}_N$ specifically applied to the final state.
iii) $\mathbf{R}$ is a positive definite weight matrix penalizing the control effort.
iv) $\mathbf{u}_i$ is the control input (thrust commands) at time step $\mathbf{i}$.
v) $\mathbf{y}_N$ represents the deviation from the terminal state, evaluated at the final step of the prediction horizon.

*1) **The Cost Vector** $\mathbf{y}_i$:* The cost vector $\mathbf{y}_i$ is computed as the difference between the actual and reference states. The components of $\mathbf{y}_i$ are based on the position, attitude, velocity, angular velocity, thrust, and other relevant parameters of the quadrotor.
Each component of the cost vector is defined as

i) **Position Error:** The position error $\mathbf{p}_{err}$ represents the deviation between the actual position $\mathbf{p}$ of the quadrotor and the desired position $\mathbf{p}_{ref}$.

$$\mathbf{p}_{err} = \mathbf{p} - \mathbf{p}_{ref}$$

ii) **Orientation (Attitude) Error:** The attitude error $\mathbf{q}_w$ is calculated as the difference between the reference quaternion $\mathbf{q}_{ref}$ and the actual quaternion $\mathbf{q}$:

$$\mathbf{q}_w = \mathbf{q}_{ref} \circ \mathbf{q}^{-1}$$

where $\circ$ denotes quaternion multiplication. This attitude error is then split into z and xy rotation according to :

$$\mathbf{q}_e = \mathbf{q}_z \circ \mathbf{q}_{xy}$$

$\mathbf{q}_{xy} = \begin{bmatrix} q_{xy,w} & q_{xy,x} & q_{xy,y} & 0 \end{bmatrix}^T$ and

$\mathbf{q}_z = \begin{bmatrix} q_{z,w} & 0 & 0 & q_{z,z} \end{bmatrix}^T$
The attitude error vector is constructed as follows:

$$\mathbf{q}_{err} = \begin{bmatrix} \mathbf{q}_{xy} - \mathbf{q}_{ref,xy} \\ \mathbf{q}_z - \mathbf{q}_{ref,z} \end{bmatrix}$$

Here, $\mathbf{q}_{xy}$ represents the 2D part (pitch and roll), and $\mathbf{q}_z$ represents the yaw.

iii) **Velocity Error:** The velocity error $\mathbf{v}_{err}$ is the difference between the actual velocity $\mathbf{v}$ and the desired reference velocity $\mathbf{v}_{ref}$.

$$\mathbf{v}_{err} = \mathbf{v} - \mathbf{v}_{ref}$$

iv) **Angular Velocity Error:** The angular velocity error $\boldsymbol{\omega}_{err}$ represents the difference between the actual angular velocity and the desired angular velocity $\boldsymbol{\omega}_{ref}$.

$$\boldsymbol{\omega}_{err} = \boldsymbol{\omega} - \boldsymbol{\omega}_{ref}$$

v) **Control Input Error:** The control input error $\mathbf{u}_{err}$ is the difference between the actual control input $\mathbf{u}$ (thrust command) and the desired reference control input $\mathbf{u}_{ref}$.

$$\mathbf{u}_{err} = \mathbf{u} - \mathbf{u}_{ref}$$

Complete Cost Vector $\mathbf{y}_i$:

$$\mathbf{y}_i = \begin{bmatrix} \mathbf{p}_{err} \\ \mathbf{q}_{xy,err} \\ \mathbf{q}_{z,err} \\ \mathbf{v}_{err} \\ \boldsymbol{\omega}_{err} \\ \mathbf{u}_{err} \end{bmatrix} = \begin{bmatrix} \mathbf{p_i} - \mathbf{p}_{ref,i} \\ \mathbf{q}_{xy,i} - \mathbf{q}_{ref,xy,i} \\ \mathbf{q}_{z,i} - \mathbf{q}_{ref,z,i} \\ \mathbf{v}_i - \mathbf{v}_{ref,i} \\ \boldsymbol{\omega}_i - \boldsymbol{\omega}_{ref,i} \\ \mathbf{u}_i - \mathbf{u}_{ref,i} \end{bmatrix}$$

Where:

i) $\mathbf{p}$: Position of the quadrotor's center of gravity expressed in the inertial frame.
ii) $\mathbf{q} = \begin{bmatrix} q_w & q_x & q_y & q_z \end{bmatrix}^T$ is the quaternion representation of the quadrotor orientation

iii) $\mathbf{v}$: Velocity of the quadrotor of the center of gravity expressed in the inertial frame
iv) $\boldsymbol{\omega} = \begin{bmatrix}\omega_x & \omega_y & \omega_z\end{bmatrix}^T$ The angular velocity of the quadrotor (body frame).
v) Control input $\mathbf{u} = \begin{bmatrix}u_1 & u_2 & u_3 & u_4\end{bmatrix}^T$

*2) **Weight Matrices**:* The weight matrices in NMPC shape the optimization process by prioritizing specific performance objectives. By carefully tuning these matrices, we can balance competing goals such as tracking accuracy, energy efficiency, and system stability.

i) **State Weights (Q)** The weight matrix Q is block-diagonal, with each block corresponding to the weights assigned to the deviations of the system's states at each time step. It is defined as:

$$\mathbf{Q} = \text{diag}(\begin{bmatrix}\mathbf{Q}_p & \mathbf{Q}_{xy} & \mathbf{Q}_z & \mathbf{Q}_v & \mathbf{Q}_\omega & \mathbf{Q}_u\end{bmatrix})$$

Where:

a) $\mathbf{Q}_p$: Penalizes position errors.
b) $\mathbf{Q}_{xy}$: Penalizes errors in roll and pitch.
c) $\mathbf{Q}_z$: Penalizes errors in yaw.
d) $\mathbf{Q}_v$: Penalizes errors in velocity.
e) $\mathbf{Q}_\omega$: Penalizes errors in angular velocity.
f) $\mathbf{Q}_u$: Penalizes control input effort .

ii) **Terminal Weights ($\mathbf{Q}_N$)** The terminal weight matrix $\mathbf{Q}_N$ emphasizes the final state of the system in the prediction horizon. A higher value for $\mathbf{Q}_N$ indicates greater importance in minimizing the deviation from the desired final state at the end of the control horizon.

iii) **Control Effort Weights (R)** The weight matrix $\mathbf{R}$ penalizes significant control inputs, promoting smooth and efficient control. Larger values of $\mathbf{R}$ reduce the magnitude of control inputs, ensuring that the quadrotor's motors operate within reasonable limits.

E. Integration of NMPC into PX4 Firmware

*1) **Overview of NMPC Integration**:* For NMPC integration, libraries such as *CasADi* (for modeling) and *Acados* (for real-time execution) are used. The generated *C*-code is embedded as a ROS node, ensuring low latency and seamless operation within the system.

*2) **Tools and Libraries deployed**:*

- **CasADi:** A symbolic framework to define dynamic models and formulate optimization problems, used to represent UAV equations of motion and constraints.
- **Acados:** A library for solving real-time nonlinear optimal control problems with fast solvers (e.g., Sequential Quadratic Programming) and *C*-code generation for embedded systems.
- **PX4-Firmware:** PX4's modular design supports integrating custom flight modes. Real-time state estimation via uORB topics allows closed-loop NMPC operation.

*3) **NMPC Problem Formulation**:*

i) **Cost Function:** NMPC minimizes trajectory deviations while ensuring stability and meeting constraints. The optimization uses predicted state trajectories and applies bounds on thrust commands.
ii) **Acados Configuration:** The NMPC problem is defined in Python using *CasADi* and exported as optimized *C*-code using *Acados*. The workflow includes:
a) **Model Definition:** Symbolic description of the cost function and constraints.
b) **Solver Settings:** The following options were used to configure solver

```
qp_solver = 'PARTIAL_CONDENSING_HPIPM'
hessian_approx = 'GAUSS_NEWTON'
nlp_solver_type = 'SQP_RTI'
integrator_type = 'ERK'
```

c) **Code Generation:** *Acados* generates solver code in *C*, compatible with the embedded platforms.

*4) **Integration with PX4**:* The integration with the PX4 autopilot is achieved through a ROS 2 node, the implementation process involves the following key steps:

i) **Data Acquisition** A custom ROS 2 subscriber node is developed to collect real-time data from PX4 topics essential for the NMPC solver. These topics include:
- *vehicle_local_position:* Provides the local position of the vehicle in the inertial frame.
- *vehicle_angular_velocity:* Supplies the vehicle's angular velocity for precise attitude control.

- *vehicle_attitude:* Delivers the current attitude (orientation) of the vehicle.

This data is structured and fed into the NMPC solver, ensuring it has the required state information to compute optimal control actions.

ii) **NMPC Solver Integration** The NMPC solver processes the received data, predicting the future behavior of the system based on the vehicle's current state and desired trajectory.

iii) **Actuator Output Generation** The computed actuator commands are in the form of control inputs required for stabilizing and maneuvering the vehicle. These outputs include thrust and torques distributed across the motors.

iv) **Command Publishing** A ROS 2 publisher node sends the calculated actuator commands to PX4 via the *actuator_motor* topic. This topic directly interfaces with the PX4 mixer, translating the control inputs into motor speeds and ensuring real-time execution of NMPC-based control strategies.

*5) **Results and Conclusion:***

i) During initial testing of the Nonlinear Model Predictive Control (NMPC) controller, it was observed that the controller faced significant challenges in achieving the desired stability. The solver encountered technical faults, preventing the controller from adequately solving for the desired outputs. These issues highlighted limitations in the robustness of the NMPC approach under the tested conditions.

ii) The initial testing and evaluation of NMPC revealed critical limitations in its ability to handle fault-tolerant control requirements for quadcopters. Consequently, we transitioned to Reinforcement Learning (RL) for fault-tolerant quadcopter control. RL offers a more adaptive and robust solution by leveraging its ability to learn from dynamic environments and handle unexpected scenarios effectively.

V. Reinforcement Learning for Fault-Tolerant Quadcopter Control [4][5]

A. *Introduction*

We propose an RL-based method for controlling a quadcopter in various operational states, including scenarios with 3 or 4 functioning motors. The approach integrates fault-tolerant and normal flight control using two neural networks: a policy network for action distribution (mean and standard deviation) and a value network for state evaluation, enabling autonomous adaptation to diverse conditions. Unlike traditional Nonlinear Model Predictive (NMP) control, this method uses machine learning to learn control policies without explicit quadcopter dynamics, framing the problem as a Markov Decision Process (MDP) solved through RL.

We implemented two deep RL algorithms, Advantage Actor-Critic (A2C) and Proximal Policy Optimization (PPO), which directly optimize policies and are better suited for continuous action spaces than Deep Q-Network (DQN). Testing was conducted with a simulated IRIS quadcopter in Gazebo's real-time physics simulator, with ROS enabling seamless integration and easy adaptation to real-world drone applications.

B. *Formulation of problem using MDP*

Our objective is to train a reinforcement learning-based agent to learn an optimal flight policy for a quadcopter with either four or three operational motors. We model this problem as an infinite-horizon Markov Decision Process (MDP), represented by the tuple (S, A, P, r, D,), where:

S: State space is the Odometry data generated using IMU and the motor feedback (the actual angular velocity of each individual motor)

$$\text{S} = \begin{bmatrix} \text{position(3)} \\ \text{linear\_velocity(3)} \\ \text{attitude(3)} \\ \text{angular\_velocity(3)} \\ \text{previous\_action(4)} \\ \text{motor\_feedback(4)} \end{bmatrix}$$

A: Action space is an n dimensional continuous space vector, a(1)...a(4) where 4 is the number of possible rotors, and a(1) to a(4) are control inputs.

P(S,A,S'): is the transition probability distribution,
r: S $\rightarrow \mathbb{R}$ is the reward function,
$\alpha, \beta, \gamma, \delta$ are scaling factors for each error term.
D(s) is the distribution of the state s_0

C. *Methodology*

1) Reinforcement Learning Algorithms: For solving the problem of learning a flight controller capable of flying a quadcopter despite losing

one or two actuators we used the reinforcement learning method mentioned in TRPO and PPO, PPO algorithm, and the approach mentioned in for successfully training a policy gradient based flight controller. Our reinforcement learning-based algorithm receives the state and produces the angular velocities of each motor directly.
we use the following conventions:

- State-action value (Q-function): The expected cumulative reward starting from state st, taking action at, and following the policy π afterward.
- Value function (V-function): The expected cumulative reward starting from state s_t and following the policy π after that
- Advantage function: Measures how much better it is to take action a_t in state s_t compared to following the policy π.
  We aim to minimize the total loss function, which combines the clipped surrogate objective LCLIP, the value function loss LVF, and an entropy bonus term S(θ). This approach is discussed in detail and further enhanced for learning a flight policy
  Where ϵ is a hyperparameter. The clip function modifies the probability ratio rt(θ) by restricting it to the range [1ϵ,1+ϵ].

$$Q_\pi(s_t, a_t) = \mathbb{E}_{s_{t+1},a_{t+1},\dots}\left[\sum_{l=0}^{\infty}\gamma^l r(s_{t+1})\right]$$

$$V_\pi(s_t) = \mathbb{E}_{a_t,s_{t+1},\dots}\left[\sum_{l=0}^{\infty}\gamma^l r(s_{t+1})\right]$$

$$A_\pi(s_t, a_t) = Q_\pi(s_t, a_t) - V_\pi(s_t)$$

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)\hat{A}_t,\ \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)\right]$$

The parameters are defined as follows:

- s_t: Current state
- a_t: Action taken in state s_t
- π: Policy, a strategy mapping states to actions.
- $\mathbb{E}$: Expectation (average) over possible outcomes.
- $r(s_{t+1})$: Reward obtained from state s_{t+1}.
- γ: Discount factor ($0 \le \gamma < 1$) that weighs the importance of future rewards.
- $Q_\pi(s_t, a_t)$: Action-Value function.
- $V_\pi(s_t)$: State-Value function.
- $A_\pi(s_t, a_t)$: Advantage function measures the relative benefit of taking action a_t compared to following policy π.

D. *ALGORITHM IMPLEMENTATION*

1) Artificial Neural Network: A neural network similar to work in, with two hidden layers of 256 neurons in each layer, is used. The policy and value networks have a similar shape except for the output layers, where the value network has one output.
2) Reward Function: We used a reward-shaping technique based on positions, orientation(quaternion), velocity, and angular velocity. If the rotors exceed a threshold in pitch and roll, we reset the position because we consider it as a terminal state. The reward function for the case of 4 motors considers the yaw angle as well in the reward calculation; however, in the case of 3 working motors, we do not consider the yaw angle in reward generation due to the necessary rotational movement of the quadcopter around the yaw axis.

VI. Simulation Setup

A. ***Simulation Configuration Overview:***

The simulation environment uses Windy World in Gazebo to evaluate the drone's behavior under various challenging conditions. This setup tests the drone's response to motor failures and environmental noise, ensuring robust performance under real-world scenarios.

B. ***Windy World in Gazebo:***

1) Environment Description: The Windy World simulation scenario introduces realistic environmental noise, such as wind disturbances and turbulence, to emulate unpredictable external conditions.

2) Failure Testing: Motor failure scenarios are simulated to observe the drone's stability, maneuverability, and control effectiveness during critical events.
3) Noise Modeling: Wind disturbances are modeled dynamically, influencing the drone's position, attitude, and angular velocity, which adds complexity to the testing environment.

C. *Simulation Control:*

The simulations are controlled through the following methods:

1) ROS Commands:
   - ROS 2 nodes manage simulation parameters and send commands to the drone in real-time.
   - Commands such as *gazebo/set_model_state* allow for dynamic adjustments, such as inducing motor failures or altering environmental conditions.
2) Python MAVSDK:
   - MAVSDK scripts provide a high-level interface to PX4 via MAVLink, enabling precise control of the drone's behavior in the simulation.
   - Python scripts are used for tasks such as arming, disarming, mission execution, and introducing specific failure scenarios.

D. *Key Simulation Features:*

1) Failure Scenarios: Simulated motor failures allow the evaluation of the drone's fault-tolerant capabilities and emergency recovery performance.
2) Noisy Conditions: The introduction of environmental noise stresses the drone's stability and highlights areas for improvement in control and recovery systems.
3) Data Logging: Comprehensive logging of drone states, actuator commands, and environmental parameters facilitates in-depth analysis and validation of the system's robustness.

This simulation setup provides a rigorous platform to test the drone's behavior under adverse conditions, focusing on its ability to maintain stability and control in the face of motor failures and environmental disturbances.
Using Gazebo's Windy World enhances the realism and reliability of the simulation results.

VII. Future Directions and Lessons Learned

Future Scope and Recommendations:

1) **Integration of Advanced Reinforcement Learning Algorithms:**
   - Extend the use of Reinforcement Learning (RL) for autonomous UAV control in dynamic and unpredictable environments. RL can adaptively handle complex scenarios like wind disturbances, obstacle avoidance, and multi-rotor coordination.
   - Explore deep RL algorithms such as Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC) to improve stability and decision-making.
2) **Real-Time Optimization of NMPC**
   - Improve the computational efficiency of the Nonlinear Model Predictive Controller (NMPC) using advanced solvers or GPU acceleration.
   - Explore adaptive NMPC, where the prediction horizon and constraints dynamically adjust based on flight conditions and mission requirements.
   - Incorporate ensemble learning models to complement the Multi-Layer Perceptron (MLP). Combining multiple algorithms (e.g., decision trees, gradient boosting) may improve classification accuracy and robustness to noisy sensor data.

Challenges faced and Lessons Learned

1) **Model Deployment Challenges:** Translating MLP and RL models into PX4-compatible formats necessitated lightweight and efficient implementations. ACADOS proved useful for NMPC code generation but required careful parameter tuning.
2) **Importance of Dataset Quality:** The accuracy of the MLP model heavily depended on the diversity and quality of the dataset. Including diverse flight conditions (e.g., various angles and motions) ensured robust failure detection.
3) **Leveraging PX4 Modularity:** Leveraging PX4's modular architecture, NMPC and custom failure detection logic were integrated seamlessly. However, configuring uORB topics

and maintaining real-time performance posed challenges.

4) **Real-Time Computation and Code Optimization:** Real-time computation in PX4's limited environment. For this, the C-code generated by *Acados* was optimized, which ensured low latency.
5) **Integrating Custom Solver into PX4's Architecture:** Integration of custom solver within PX4's architecture. The modular design of PX4 allowed smooth integration as a custom mode.
6) **Tuning Model Parameters for Optimal Performance:** Tuning weights ($\mathbf{Q}$, $\mathbf{R}$) for stable performance. To overcome this iterative fine-tuning during simulations was done.

References

[1] A. Dutta, J. Wang, F. Kopsaftopoulos, and F. Gandhi, "Rotor fault detection and identification on multicopter based on statistical data-driven methods: Experimental assessment via flight tests," May 2022, pp. 1–14. DOI: 10.4050/F-0078-2022-17556.

[2] A. Dutta, J. Wang, F. Kopsaftopoulos, and F. Gandhi, "Rotor fault detection and identification on multicopter based on statistical data-driven methods: Experimental assessment via flight tests," in *Proceedings of the Vertical Flight Society's 78th Annual Forum & Technology Display*, PhD Student (Dutta), Undergraduate Student (Wang), Assistant Professor (Kopsaftopoulos), Redfern Professor (Gandhi), Center for Mobility with Vertical Lift (MOVE), Rensselaer Polytechnic Institute, Ft. Worth, Texas, USA: Vertical Flight Society, May 2022.

[3] F. Nan, S. Sun, P. Foehn, and D. Scaramuzza, "Nonlinear mpc for quadrotor fault-tolerant control," *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 5047–5054, 2022. DOI: 10.1109/LRA.2022.3153841.

[4] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[5] W. Koch, R. Mancuso, R. West, and A. Bestavros, "Reinforcement learning for uav attitude control," *ACM Transactions on Cyber-Physical Systems*, vol. 3, Apr. 2018. DOI: 10.1145/3301273.